27th International Conference on Principles of Distributed Systems

OPODIS 2023, December 6–8, 2023, Tokyo, Japan

Edited by

Alysson Bessani
Xavier Défago
Junya Nakamura
Koichi Wada
Yukiko Yamauchi

LIPIcs – Vol. 286 – OPODIS 2023 www.dagstuhl.de/lipics

Editors

Alysson Bessani
University of Lisbon, Portugal
anbessani@fc.ul.pt

Xavier Défago
Tokyo Institute of Technology, Japan
defago@c.titech.ac.jp

Junya Nakamura
Toyohashi University of Technology, Japan
junya@imc.tut.ac.jp

Koichi Wada
Hosei University, Japan
wada@hosei.ac.jp

Yukiko Yamauchi
Kyushu University, Japan
yamauchi@inf.kyushu-u.ac.jp

ACM Classification 2012
Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms; Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis; Networks → Mobile networks; Networks → Wireless access networks; Networks → Ad hoc networks; Computing methodologies → Distributed algorithms; Security and privacy → Distributed systems security; Information systems → Distributed storage; Computer systems organization → Dependable and fault-tolerant systems and networks; Software and its engineering → Distributed systems organizing principles

ISBN 978-3-95977-308-9

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/978-3-95977-308-9.

Publication date
January, 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.OPODIS.2023.0
ISBN 978-3-95977-308-9 **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

Invited Talks

Regular Papers

27th International Conference on Principles of Distributed Systems (OPODIS 2023).
Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The papers in this volume were presented at the 27th International Conference on Principles of Distributed Systems (OPODIS 2023), held on December 6–8, 2023 in Tokyo, Japan.

OPODIS is an open forum for the exchange of state-of-the-art knowledge about distributed computing. With strong roots in the theory of distributed systems, OPODIS has expanded its scope to cover the entire range between the theoretical aspects and practical implementations of distributed systems, as well as experimental and quantitative assessments. All aspects of distributed systems are within the scope of OPODIS: theory, specification, design, performance, and system building. Specifically, this year, the topics of interest at OPODIS included:

- Big data analytics frameworks
- Blockchain, theory and practice
- Cloud, grid, edge computing
- Communication and mobile networks
- Data centers
- Data-intensive computing
- Dependable and secure systems
- Distributed file systems and database systems
- Distributed graph algorithms
- Distributed recommender systems
- Distributed storage
- Distributed systems for machine learning
- Fault tolerance and consistency
- Game-theory in distributed computing
- Impossibility results in distributed computing
- IoT
- Middleware and Operating systems
- Mobile agents and robots
- Parallelism, concurrency, and multicore systems
- Self-stabilizing, self-organizing and autonomous systems
- Shared and transactional memory, memory management

We received 96 submissions, each of which underwent a double-blind peer review process except one submission retracted just after the deadline and two desk rejected due to anonymity issues. The submissions were reviewed by at least three Program Committee (PC) members, with the help of external reviewers, and were extensively discussed by the PC members. In the end, 32 papers were selected to be included in these proceedings. Overall, the quality of the submissions was very high, and many interesting contributions could not be accepted due to the physical limitations of the conference.

The PC also voted for selecting the best paper and best student paper in the program. The best paper award was given to "*Fault-tolerant computing with unreliable channels*" by A. Pastoriza, G. Chockler, and A. Gotsman; and the best student paper award was given to "*Flooding with Absorption: An Efficient Protocol for Heterogeneous Bandits over Complex Networks*" by J. Lee, L. Schmid, and S. Yun.

The OPODIS proceedings appear in the Leibniz International Proceedings in Informatics (LIPIcs) series. LIPIcs proceedings are available online and free of charge to readers. The production costs are paid in part from the conference budget.

27th International Conference on Principles of Distributed Systems (OPODIS 2023).
Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This year OPODIS had three distinguished invited keynote speakers: Vincent Gramoli (University of Sydney and Redbelly Network), François Le Gall (Nagoya University), and Roger Wattenhofer (ETH Zurich).

Thank you to all the authors that submitted their work to OPODIS. We are also grateful to the Program Committee members for their hard work reviewing papers and their active participation in the online discussions. We also thank the external reviewers for their help with the reviewing process.

Organizing this event would not have been possible without the help of the organizing committee: François Bonnet (Tokyo Institute of Technology), Yuichi Sudo (Hosei University), Mayuko Takano (Tokyo Institute of Technology), Yasumasa Tamura (Tokyo Institute of Technology).

We are also grateful for the support we received from our students at Tokyo Institute of Technology and at Hosei University: Yusuke Ichiki, Tinghong Jin, Yuuki Kin, Genta Konno, Keita Nakajima, Gimpei Nakase, Bao-Ngoc Nguyen, Justin Shetty, Ryo Yui (Tokyo Institute of Technology), and Naomasa Kohara, Kaito Takase (Hosei University).

Finally, we thank the Steering Committee members for their valuable advice and would like to express our gratitude to our sponsors, National Institute of Information and Communications Technology (NICT), Institute of Electronics, Information and Communication Engineers (IEICE), and Information Processing Society of Japan (IPSJ) for their generous support.

November 2023

Alysson Bessani (Faculdade de Ciências, Universidade de Lisboa, Portugal)
Xavier Défago (Tokyo Institute of Technology, Japan)
Junya Nakamura (Toyohashi University of Technology, Japan)
Koichi Wada, (Hosei University, Japan)
Yukiko Yamauchi (Kyushu University, Japan)

- *Supported by International Exchange Program of National Institute of Information and Communications Technology (NICT).*
- *Sponsored by the Institute of Electronics, Information and Communication Engineers (IEICE), Japan.*
- *Sponsored by Information Processing Society of Japan (IPSJ).*

Program Committee

General Chair

Xavier Défago, Tokyo Institute of Technology, Japan
Koichi Wada, Hosei University, Japan

Publication Chair

Junya Nakamura, Toyohashi University of Technology, Japan

Program Chairs

Alysson Bessani, Faculdade de Ciências, Universidade de Lisboa, Portugal
Yukiko Yamauchi, Kyushu University, Japan

Program Committee

Eduardo Alchieri, University of Brasilia, Brazil
Anish Arora, The Ohio State University, USA
Hagit Attiya, Technion, Israel
François Bonnet, Tokyo Institute of Technology, Japan
Silvia Bonomi, Sapienza University of Rome, Italy
Shantanu Das, Aix-Marseille University, France
Joshua J. Daymude, Arizona State University, USA
Tobias Distler, University of Erlangen-Nuremberg, Germany
Sisi Duan, Tsinghua University, China
Yuval Emek, Technion, Israel
Bernardo Ferreira, Faculdade de Ciências, Universidade de Lisboa, Portugal
Paola Flocchini, University of Ottawa, Canada
Leszek A. Gąsieniec, University of Liverpool, UK
Seth Gilbert, National University of Singapore, Singapore
Magnus M. Halldorsson, Reykjavik University, Iceland
Eshcar Hillel, Pliops, Israel
Taisuke Izumi, Osaka University, Japan
Sayaka Kamei, Hiroshima University, Japan
Yonghwan Kim, Nagoya Institute of Technology, Japan
Evangelos Kranakis, Carleton University, Canada
Fabian Kuhn, University of Freiburg, Germany
Anissa Lamani, University of Strasbourg, France
Euripides Markou, University of Thessaly, Greece
Achour Mostéfaoui, University of Nantes, France
Junya Nakamura, Toyohashi University of Technology, Japan
Roberto Palmieri, Lehigh University, USA
Fernando Pedone, University of Lugano, Switzerland
Maria Potop-Butucaru, Sorbonne University, France
Nuno Preguiça, University NOVA Lisboa, Portugal
Vincent Rahli, University of Birmingham, UK
Étienne Rivière, UCLouvain, Belgium
Christian Scheideler, Paderborn University, Germany

27th International Conference on Principles of Distributed Systems (OPODIS 2023).
Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Valerio Schiavoni, University of Neuchâtel, Switzerland
Gregory Schwartzman, Japan Advanced Institute of Science and Technology, Japan
Jukka Suomela, Aalto University, Finland
Sébastien Tixeuil, Sorbonne University, France

Steering Committee

Panagiota Fatourou, University of Crete, Greece
Pascal Felber, University of Neuchâtel, Switzerland **(Chair)**
Paola Flocchini, University of Ottawa, Canada
Vincent Gramoli, University of Sydney and EPFL, Australia
Yannic Maus, TU Graz, Austria
Alessia Milani, LIS, Aix-Marseille Université, France
Rotem Oshman, Tel-Aviv University, Israel
Paolo Romano, INESC-ID, University of Lisbon, Portugal

27th International Conference on Principles of Distributed Systems (OPODIS 2023).
Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Reviewers

Evangelos Bampas, Université Paris-Saclay
Rida A. Bazzi, Arizona State University
Armando Castañeda, Instituto de Matemáticas UNAM, México
Jérémie Chalopin, CNRS, Aix-Marseille Université
Yi-Jun Chang, National University of Singapore
Bapi Chatterjee, Indraprastha Institute of Information Technology
Ryota Eguchi, Nara Institute of Science and Technology
Maxime Flin, Reykjavik University
Konstantinos Georgiou, Toronto Metropolita University
Nikos Giachoudis, Foundation for Research and Technology - Hellas (FORTH)
Thorsten Götte, Paderborn University
Ahmed Hassan, Lehigh University
David Ilcinkas, CNRS, Université de Bordeaux
Hirotsugu Kakugawa, Ryukoku University
Giorgos Kappes, University of Ioannina
Naoki Kitamura, Osaka University
Maria Kokkou, Aix-Marseille University
Arnaud Labourel, LIS, Aix-Marseille University
David Liedtke, Paderborn University
Henrik Lievonen, Aalto University
Susumu Nishimura, Kyoto University
Alexandre Nolin, CISPA - Helmholtz Center for Information Security
Dennis Olivetti, Gran Sasso Science Institute and Institut de Recherche en Informatique Fondamentale
Fukuhito Ooshita, Fukui University of Technology
Aris Pagourtzis, National Technical University of Athens
Ami Paz, LISN - CNRS & Paris Saclay University
Arie Poran, Ono Academic College
Sergio Rajsbaum, UNAM
Dror Rawitz, Bar Ilan University
Masahiro Shibata, Kyushu Institute of Technology
Yuichi Sudo, Hosei University
Ahmed Wade, École Polytechnique de Thiès
Daniel Warner, University of Paderborn

27th International Conference on Principles of Distributed Systems (OPODIS 2023).
Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

From Consensus Research to Redbelly Network Pty Ltd

Vincent Gramoli
University of Sydney, Australia
Redbelly Network, Sydney, Australia

Abstract

Designing and implementing correctly a blockchain system requires collaborations across places and research fields. Redbelly, a company across Australia, India and USA, illustrates well this idea.

It started in 2005 at OPODIS, where we published the Reconfigurable Distributed Storage to replace distributed participants offering a service without disrupting its availability. This line of work [5] was instrumental to reconfigure blockchains without introducing hard forks. The research on the consensus problem we initiated at IRISA [4] led to rethinking PBFT-like algorithms for the context of blockchain by getting rid of the leader that can act as the bottleneck of large networks [6]. Our work on security led to disclosing vulnerabilities in Ethereum [3] and then motivated us to formally verify blockchain consensus [1]. Our work at the frontier of economics [9] led us to prevent front-running attacks [11] and to incentivize rational players to behave [8]. Our system work at Cornell and then at EPFL was foundational in experimenting blockchains across the globe [7].

Although not anticipated at the time, this series of work progressively led the University of Sydney and CSIRO, and later Redbelly Network Pty Ltd, to design the Redbelly Blockchain [2, 10], the platform of choice for compliant asset tokenisation.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Innovations, Commercialisation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.1

Category Invited Talk

Funding This work is supported by the ARC Future Fellowship (#180100496).

References

1 Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazic, Pierre Tholoniat, and Josef Widder. Holistic verification of blockchain consensus. In *DISC*, 2022. doi:10.4230/LIPICS.DISC.2022.10.

2 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red Belly: A secure, fair and scalable open blockchain. In *S&P*, pages 466–483. IEEE, 2021. doi:10.1109/SP40001.2021.00087.

3 Parinya Ekparinya, Vincent Gramoli, and Guillaume Jourjon. The attack of the clones against proof-of-authority. In *NDSS*, 2020. URL: https://www.ndss-symposium.org/ndss-paper/the-attack-of-the-clones-against-proof-of-authority/.

4 V. Gramoli. *Blockchain Scalability and its Foundations in Distributed Systems*. Springer, 2022. doi:10.1007/978-3-031-12578-2.

5 V. Gramoli, N. Nicolaou, and A. A. Schwarzmann. *Consistent Distributed Storage*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2021. doi:10.2200/S01069ED1V01Y202012DCT017.

6 V. Gramoli and Q. Tang. The future of blockchain consensus. *CACM*, 66(7):79–80, 2023. doi:10.1145/3589225.

7 Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. Diablo: A benchmark suite for blockchains. In *EuroSys*, pages 540–556. ACM, 2023. doi:10.1145/3552326.3567482.

8 Alejandro Ranchal-Pedrosa and Vincent Gramoli. TRAP: the bait of rational players to solve byzantine consensus. In *AsiaCCS*, pages 168–181. ACM, 2022. doi:10.1145/3488932.3517386.

9 Michael Spain, Sean Foley, and Vincent Gramoli. The impact of Ethereum throughput and fees on transaction latency during ICOs. In *Tokenomics*, 2019. doi:10.4230/OASICS.TOKENOMICS.2019.9.

10 Deepal Tennakoon, Yiding Hua, and Vincent Gramoli. Smart Redbelly blockchain: Reducing congestion for Web3. In *IPDPS*, pages 940–950. IEEE, 2023. doi:10.1109/IPDPS54959.2023.00098.

11 Pouriya Zarbafian and Vincent Gramoli. Lyra: Fast and scalable resilience to reordering attacks in blockchains. In *IPDPS*, pages 929–939. IEEE, 2023. doi:10.1109/IPDPS54959.2023.00097.

Quantum Distributed Computing: Potential and Limitations

François Le Gall
Graduate School of Mathematics, Nagoya University, Japan

Abstract

The subject of this talk is quantum distributed computing, i.e., distributed computing where the processors of the network can exchange quantum messages. In the first part of the talk I survey recent results [3, 4, 5, 6, 8] and some older results [1, 7] that show the potential of quantum distributed algorithms. In the second part I present our recent work [2] showing the limitations of quantum distributed algorithms for approximate graph coloring. Finally, I mention interesting and important open questions in quantum distributed computing.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Quantum computation theory

Keywords and phrases Quantum computing, distributed algorithms, CONGEST model, LOCAL model

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.2

Category Invited Talk

References

1 Michael Ben-Or and Avinatan Hassidim. Fast quantum byzantine agreement. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 481–485, 2005. doi:10.1145/1060590.1060662.

2 Xavier Coiteux-Roy, Francesco d'Amore, Rishikesh Gajjala, Fabian Kuhn, François Le Gall, Henrik Lievonen, Augusto Modanese, Marc-Olivier Renou, Gustav Schmid, and Jukka Suomela. No distributed quantum advantage for approximate graph coloring. *CoRR*, abs/2307.09444, 2023. doi:10.48550/ARXIV.2307.09444.

3 Taisuke Izumi and François Le Gall. Quantum distributed algorithm for the All-Pairs Shortest Path problem in the CONGEST-CLIQUE model. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 84–93, 2019. doi:10.1145/3293611.3331628.

4 Taisuke Izumi, François Le Gall, and Frédéric Magniez. Quantum distributed algorithm for triangle finding in the CONGEST model. In *Proceedings of the 37th International Symposium on Theoretical Aspects of Computer Science (STACS 2020)*, pages 23:1–23:13, 2020. doi:10.4230/LIPICS.STACS.2020.23.

5 François Le Gall and Frédéric Magniez. Sublinear-time quantum computation of the diameter in CONGEST networks. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC 2018)*, pages 337–346, 2018. doi:10.1145/3212734.3212744.

6 François Le Gall, Harumichi Nishimura, and Ansis Rosmanis. Quantum advantage for the LOCAL model in distributed computing. In *Proceedings of the 36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, pages 49:1–49:14, 2019. doi:10.4230/LIPICS.STACS.2019.49.

7 Seiichiro Tani, Hirotada Kobayashi, and Keiji Matsumoto. Exact quantum algorithms for the leader election problem. *ACM Transactions on Computation Theory*, 4(1):1:1–1:24, 2012. doi:10.1145/2141938.2141939.

8 Xudong Wu and Penghui Yao. Quantum complexity of weighted diameter and radius in CONGEST networks. In *Proceedings of the 42nd ACM Symposium on Principles of Distributed Computing (PODC 2022)*, pages 120–130, 2022. doi:10.1145/3519270.3538441.

Distributed Algorithms as a Gateway To Deductive Learning

Roger Wattenhofer
ETH Zurich, Switzerland

Abstract

With the book Thinking Fast and Slow, Daniel Kahneman popularized the idea that the human brain can think in two different modes. The fast mode is instinctive and automatic, while the slow mode is deliberative and logical. As of 2023, one can argue that machine learning understands how to think fast. Deep neural networks are remarkably successful in rapidly classifying and regressing data. Thinking slow on the other hand is still a mystery. Large language models may provide an illusion of being able to think slow. However, prompts that need multiple deductive steps are generally beyond the capabilities of large language models. Distributed algorithms have the potential to help understanding deductive reasoning. Distributed algorithms usually consist of several little steps, iteratively applied, each step being easily learnable. As such distributed computing may provide an interesting bridge towards understanding deduction, extrapolation, reasoning, and everything else needed to think slow. In the talk, we will discuss some exciting case studies from graph generation to origami folding.

2012 ACM Subject Classification Theory of computation → Machine learning theory; Computing methodologies → Distributed algorithms

Keywords and phrases abstract visual reasoning, agent-based reasoning, classic algorithm benchmarks, differentiable status registers, explainable graphs, graph generation algorithms, integer sequences, neural combinatorial circuits, recurrent network algorithms, origami folding, Tatham's puzzles

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.3

Category Invited Talk

27th International Conference on Principles of Distributed Systems (OPODIS 2023).
Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi; Article No. 3; pp. 3:1–3:1
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The Synchronization Power of Auditable Registers

Hagit Attiya
Technion, Haifa, Israel

Antonella Del Pozzo
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Alessia Milani
Laboratoire d'Informatique et Systèmes, Aix-Marseille Université and CNRS, Marseille, France

Ulysse Pavloff
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Alexandre Rapetti
Aix-Marseille Université, Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract

Auditability allows to track all the read operations performed on a register. It abstracts the need of data owners to control access to their data, tracking who read which information. This work considers possible formalizations of auditing and their ramification for the possibility of providing it.

The natural definition is to require a linearization of all write, read and audit operations together (*atomic* auditing). The paper shows that atomic auditing is a powerful tool, *as it can be used to solve consensus*. The number of processes that can solve consensus using atomic audit depends on the number of processes that can read or audit the register. If there is a single reader or a single auditor (the writer), then consensus can be solved among two processes. If multiple readers and auditors are possible, then consensus can be solved among the same number of processes. This means that strong synchronization primitives are needed to support atomic auditing.

We give implementations of atomic audit when there are either multiple readers or multiple auditors (but not both) using primitives with consensus number 2 (swap and fetch&add). When there are multiple readers *and* multiple auditors, the implementation uses compare&swap.

These findings motivate a weaker definition, in which audit operations are not linearized together with read and write operations (*regular* auditing). We prove that regular auditing can be implemented from ordinary reads and writes on atomic registers.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies

Keywords and phrases Auditability, atomic register, fault tolerance, consensus number

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.4

Related Version *Full Version*: https://arxiv.org/abs/2308.04646

Funding *Hagit Attiya*: Partially supported by the Israel Science Foundation (grant 22/1425).
Alessia Milani: Partially supported by CNRS-INS2I, Project PRIDE 2023.
Alexandre Rapetti: Partially supported by CNRS-INS2I, Project PRIDE 2023.

1 Introduction

Outsourcing storage capabilities to third-party distributed storage is a common practice for both private and professional users. It helps to circumvent local space constraints, dependability, and accessibility limitations. Unfortunately, this means having to trust the distributed storage provider on data integrity, retrievability, and confidentiality. Those issues are underscored by relentless attacks on data storage servers [2], which increased the awareness to data confidentiality and sovereignty and lead to a recent worldwide deployment

of data protection regulations [1, 10, 19]. Even in secure storage systems where *access control* policies regulate who can access the data, an unauthorized user can access data either due to a misconfiguration of the access control system or in the occurrence of a data breach [17, 24].

As a result, data owners are increasingly concerned about who accesses their data. This makes *auditability* – the systematic tracking of who has read data in storage systems and the information it has observed – an important feature.

A *register* is an abstraction for distributed storage that provides read and write operations to the clients. An *auditable register*, introduced by Cogo and Bessani [5], is a register enriched with an *audit* operation. An audit operation indicates who performed read operations on it, and which values they have read. Auditability is defined in terms of two properties: *completeness* ensures that readers that access data are detected, while *accuracy* ensures that readers who do not access data are not incriminated.

In this work, we formalize the correctness of auditable registers in terms of their high-level operations (read, write and audit). We first formalize a natural extension of an atomic register, called *atomic register with atomic audit* where all the operations (including audit) appear to happen in a sequential order that respects their real-time order.

We show that an *atomic register with atomic audit* is a powerful abstraction, *because it has a greater consensus number than an ordinary atomic register*. Recall that the *consensus number* [12] of object type X is m if m is the largest integer such that there exists an asynchronous consensus algorithm for m processes, up to $m-1$ may crash, using only shared objects of type X and read/write registers.

We present a wait-free algorithm that solves consensus among *two processes*, using an atomic register with atomic audit where only one process (the writer) can perform audit operations. This stands in contrast to the well-known result [12] that atomic read/write registers cannot be used to solve wait-free consensus among two processes. We then show that when m processes can read and audit the register, it is possible to solve consensus among m processes.

These results indicate that base objects stronger than read/write registers are needed to implement atomic audit, motivating our implementations of an atomic register with atomic audit. When there is either a single auditor or a single reader, we use base objects with consensus number 2 (*swap* and *fetch&add*).

Specifically, we first present a simple algorithm for a single-reader atomic auditable register with atomic audit, where the writer is the only process that can execute the audit operations. The writer needs to atomically retrieve who read the previously written value while writing a new value. With a single reader, this can be easily ensured by using *swap* primitives: to read a value, the reader atomically swaps it with a special value. If the writer retrieves this special value when writing a new value, then it is aware of the value read.

Extending this idea to multi-readers is challenging, since readers might be swapping each other's values. We propose a solution that uses a single shared object accessed with *swap* and *fetch&add* primitives. The n low-order bits of the value stored in this object, where n is the number of readers, are used to indicate which readers have accessed the value stored in the high-order bits. Each reader is assigned a unique bit, which is set to 1 when the reader accesses the value. Then, when the writer writes a new value it can learn who read its previous value by checking the values of the low-order bits it retrieved from the value atomically read while writing the new value. A similar algorithm allows to support multiple auditors, but only a single reader, also using *swap* and *fetch&add*.

When there are multiple readers and auditors, we use *compare&swap*, which has consensus number ∞, in addition to *fetch&add*.

Taken together, our results mean that atomic audit cannot be implemented using only reads and writes, and stronger primitives (with consensus number > 1) should be used.

We then investigate the possibility to extend an atomic register with a useful audit operation without relying on strong synchronization primitives. This weaker abstraction is called a *regular* audit, and roughly speaking, differs from the previous one in not having the audit operations linearized together with read and write operations. In particular, a regular audit operation *aop* may not detect a read operation that is concurrent with it, even though the read has to be linearized before a write that completes before the invocation of *aop*. Our final result is a single-writer multi-reader atomic register with multi-auditor *regular* audit, using only atomic read and write operations, whose consensus number is 1.

Related Work. To the best of our knowledge, only two papers [5, 6] formally study auditability of read operations. Cogo and Bessani [5] were the first to formalize the notion of auditable register. Their definition is tailored for auditable register implementations on top of a shared memory model where some base objects can be faulty, i.e., they can omit to record readers or they can record nonexistent read operations. They present an algorithm to implement an auditable register, here read and write operations provide regular semantics using $n \geq 4f + 1$ atomic read/write shared objects, f of which may be faulty. Because of their failure model, their high-level register implementation relies on information dispersal schemes, where the input of a high-level write is split into several pieces each written in a different low-level shared object. It implies that a process can read a written value only if it collects enough pieces of information, making its read operation detectable. Their definition of completeness and accuracy for the auditable register relies on the notion of *effectively read*, which they formalize to capture the fact that the process executing the high-level read operation could have collected enough pieces of information and be able to retrieve the value, even if the read operation does not return.

In asynchronous message-passing systems where f processes can be Byzantine, Del Pozzo et al. [6] study the possibility of implementing an atomic auditable register with the accuracy and completeness properties, as defined by Cogo and Bessani, with fewer than $4f + 1$ servers. They prove that without communication between servers, auditability requires at least $4f + 1$ servers, f of which may be Byzantine. They also show that allowing servers to communicate with each other admits an auditable atomic register with optimal resilience of $3f + 1$. Their implementation also uses information dispersal scheme to deal with Byzantine processes. In contrast, we consider a classical shared memory model where processes fail by crashing. Also, our definition of auditable register is not tailored for a specific class of implementations, since it is stated in terms of high-level operations.

Most of the other works on auditing protocols for distributed storage focus on data integrity [7, 8, 13, 15, 16, 21–23], while our work focuses on auditing *who* has read *which* data.

When faulty processes are malicious, *accountability* [3, 4, 11, 20] aims to produce proofs of misbehavior in instances where processes deviate, in an observable way, from the prescribed protocol. This allows the identification and removal of malicious processes from the system as a way to clean the system after a safety violation. In contrast, auditability logs the processes actions and let the auditor derive conclusions on the processes behavior.

Frey, Gestin and Raynal [9] investigate the synchronization power of *AllowList* and *DenyList*: intricate append-only lists where AllowList contains resources that processes can access, while DenyList includes resources that processes cannot access. They prove the consensus number of AllowList is 1, while the consensus number of DenyList is equal to the number of processes that can access resources not listed in the DenyList. AllowList and DenyList control accesses, while an auditable register tracks (read) accesses; further discussion of the relation between an auditable register and DenyList appears in Section 8.

2 Model

We consider a standard shared-memory model where crash-prone asynchronous processes communicate through registers, using a given set of primitive operations. The primitive operations (sometimes called just primitives) include ordinary *read* and *write*, as well as *swap*, *fetch&add* and *compare&swap*.

A $swap(v)$ primitive atomically writes v to the register and returns its previous value. A $fetch\&add(a)$ primitive atomically writes the sum of a and the current value of the register into the register and returns its previous value. A $compare\&swap(old, new)$ primitive is an atomic conditional write: the write of *new* is executed if and only if the value of the register is *old*; a Boolean value is returned that indicates if the write was successful or not.

The auditable atomic register is an extension of an ordinary atomic read/write register [14]. It is formally defined in the next section. We only consider *single-writer* registers, where each register can be written by a single process.

An auditable register implementation specifies the state representation of the register and the algorithms processes follow when they perform the read, write and audit operations. Each operation has an invocation and a response event.

An execution is a sequence of steps performed by processes as they follow their algorithms, in each of which a process applies at most a single primitive to the shared memory (possibly in addition to some local computation).

A *history* H is a sequence of invocation and response events; no two events occur at the same time. An operation is *complete* in history H, if H contains both the invocation and the matching response for this operation. If the matching response is missing, the operation is *pending*. An operation *op precedes* another operation op' in H if the response of *op* appears before the invocation of op' in H; we also say that op' *follows op*.

A history is *sequential* if each operation invocation is immediately followed by the matching response, by the same process on the same object. For a given history H, $\mathsf{complete}(H)$ is the set of histories obtained from H by appending zero or more responses to some pending invocations and discarding the remaining pending invocations.

We consider *wait-free* implementations which ensures that a non faulty process completes an operation within a finite number of its own steps.

3 Definitions of Auditable Register

An auditable register supports three operations: $R.write(v)$ which assigns value v to the register R, $R.read()$ which returns the value of the register, and $R.audit()$ which reports the set of all values read in the register and by whom. Specifically, an audit operation returns a set of pairs (p, v), each corresponding to a read invoked by process p that returned v. In the following, we define two specifications for *audit* operations, exploring different semantics of their interaction with concurrent read and write operations.

Intuitively, *atomic audit* provides the illusion that all the read, write, and audit operations appear as if they have been executed sequentially.

▶ **Definition 1** (Atomic audit). *A history H is* atomic with atomic audit *if there is a history H' in $\mathsf{complete}(H)$ and a sequential history π that contains all operations in H' such that:*

1. *If operation op_1 precedes operation op_2 in H, then op_1 appears before op_2 in π.* Informally, π respects the real-time order of non-overlapping operations.
2. *Every read in π returns the value of the most recent preceding write, if there is one, and the initial value, otherwise.* Informally, the history π respects the semantics of an atomic read / write register.

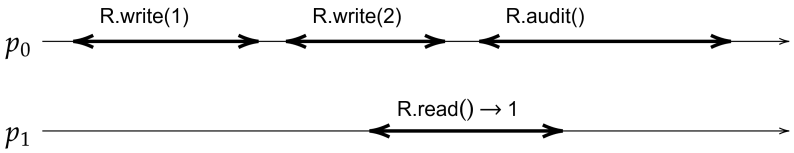


Figure 1 A scenario where a regular audit can return either $\emptyset$ or $(p_1, 1)$, while an atomic audit must return $(p_1, 1)$.

3. *Every audit op in π returns a set of pairs $\mathcal{P}$ such that*
   (π-Completeness): For each read operation op' by process p that precedes op in π, $(p, v) \in \mathcal{P}$, where v is the value returned by op'.
   (π-Accuracy): For any pair $(p, v) \in \mathcal{P}$, there is a read operation op' by process p that returned v, and op' precedes op in π.

Roughly speaking, π-*Completeness* formalizes that any read of a value from the register must be detected by the audit operation, while π-*Accuracy* ensures that a read is reported by an audit operation only if it has occurred. Note that taken together, π-Completeness and π-Accuracy say that a pair (p, v) is returned by the audit operation *if and only if* a read operation by process p, returning v, is linearized in π before the audit. That is, an *atomic audit* operation detects all the read operations linearized before it and does not detect any read operation linearized after it.

A *regular audit* operation detects all read operations that complete before the audit starts and does not detect any read operation that starts after it completes. An audit operation may detect some subset of the read operations that overlap it.

▶ **Definition 2** (Regular audit). *A history H is* atomic with regular audit *if there is a history H' in* complete(H)*, and a sequential history π that contains all read and writes operations in H' that satisfies the first two conditions of Definition 1, and in addition:*

3. *Every audit $op \in H'$ returns a set of pairs $\mathcal{P}$ such that*
   (H'-Completeness): For each read operation op' in H' by process p, that completes in H' before the invocation of op in H', $(p, v) \in \mathcal{P}$, where v is the value returned by op'.
   (H'-Accuracy): For any pair $(p, v) \in \mathcal{P}$, there is a read operation $op' \in H'$ by process p that returned v, and the invocation of op' in H' precedes the response of op in H'.

Note that while the condition on atomic audit operations (Definition 1) is stated relative to the linearization (sequential execution) π, the condition on regular audit is stated relative to the completion H' of the original history H. As we shall see, this seemingly-minor change leads to an important difference in the synchronization power of audit operations.

Figure 1 depicts a scenario where the responses of atomic audit and regular audit may differ.

In the rest of the paper, we consider that only one process can invoke write operations on the register, called the *writer*, which is also allowed to invoke audit operations. Thus, the *writer* is also an *auditor* of the register. When several processes are allowed to read the register, we call it *multi reader*; otherwise, it is *single reader*. Similarly, if several processes other than the writer can audit the register, we call it *multi auditor*; otherwise, it is *single auditor*.

For the correctness proof of the algorithms implementing the auditable registers, we assume that the values written to the register are unique.

Algorithm 1 Two-process consensus using swsr atomic registers with single-auditor atomic audit.

Shared Variables:
R_i, $i \in [0,1]$, swsr atomic register with writer / auditor p_i and reader p_{1-i}, initially $\perp$

Local Variables:
val, initially $\perp$ ▷ value read from R_{1-i}
audit_response, initially $\emptyset$ ▷ response of audit on R_i

```
1: propose(v_i)                                  ▷ Pseudo code for process p_i, i ∈ [0,1]
2:     R_i.write(v_i)
3:     val ← R_{1−i}.read()
4:     audit_response ← R_i.audit()
5:     if val =⊥
6:         return v_i
7:     if audit_response = (p_{1−i}, ⊥)
8:         return val
9:     return max(v_i, val)
```

4 Using atomic audit to solve consensus

In this section, we investigate how atomic audit allows to solve consensus. An algorithm solving consensus satisfies the following properties:

Termination: A process decides within a finite number of its own steps.

Agreement: All processes decide on the same value.

Validity: The decision value has been proposed by some process.

4.1 Single-reader register with single-auditor atomic audit solves two-process consensus

Algorithm 1 solves consensus between two processes using two single-writer single-reader (swsr) atomic registers with a single-auditor atomic audit: R_i, for each $i \in \{0,1\}$, is a swsr register written and audited by process p_i and read by p_{1-i}.

Each process first writes the value it proposes in its own register. Then it reads the other process's register and audits its own register. Finally, it returns its own value or the other process's value, accordingly to the values returned from the read and audit operations. In particular, p_i returns its own value (Line 6) if it read the initial value from R_{1-i} (Line 5). In that case, p_{1-i} reads v_i from R_i (Line 3). The condition in Line 5 would not hold, and since the audit operation on R_{1-i} detects that p_i read $\perp$ from R_{1-i} (Line 4), p_{1-i} returns the value of *val* (Line 8), which is v_i. Finally, if p_i and p_{1-i} both read the input of the other process and they know this fact thanks to the result of the audit operation, they apply a deterministic rule to break the tie and choose the same value.

The pseudocode appears in Algorithm 1. In in the full version, we prove:

▶ **Theorem 3.** *Algorithm 1 solves consensus for two processes.*

4.2 Multi-reader register with multi-auditor atomic audit solves n-process consensus

We now generalize Algorithm 1 to solve consensus among n processes using single-writer multi-reader (swmr) atomic registers with *multi-auditor* atomic audit. Like the algorithm for two-process consensus, processes leverage the audit to reconstruct at which point of the execution the other processes are, and base their decision on it.

Algorithm 2 uses n swmr atomic registers with multi-auditor atomic audit $R_0, \ldots, R_{n-1}$, all initially $\perp$. Process p_i is the single writer of R_i, and all processes can read and audit R_i.

Each process p_i proposes its input, by writing it in R_i. Then, p_i reads and audits all the other registers. A simple situation is when one process, say p_i, writes and reads before all other processes, the audit detects that p_i read v_i in R_i and $\perp$ in all other registers. This implies that all later processes will read p_i's value in R_i and, thanks to the audit, detect that p_i is not aware of the other processes' propositions. In this case, v_i is the only value known to all processes, so it is safe to decide on v_i. In general, we can consider the set P of processes that write before any process reads. No process reads $\perp$ from the registers of processes in P, and this can be detected by auditing these registers. This means that all processes consider the input values of processes in P as safe to decide upon, and agreement can be reached by deterministically picking one of these values, e.g., the maximal one.

Each process keeps the following local data structures: *values*[ ] is an array of length n to hold the values read from $R_0, \ldots, R_{n-1}$, initially $\perp$; *safe_values* is a set that stores the proposed values that no process missed, initially $\emptyset$; and *audit_response* holds the results of audits on $R_0, \ldots, R_{n-1}$, initially $\emptyset$.

When proposing a value v_i, each process p_i first writes v_i in R_i (Line 2). Next, p_i reads $R_0, \ldots, R_{n-1}$ and stores the responses in *values*[ ] (Line 4). Finally, p_i audits $R_0, \ldots, R_{n-1}$ and stores the returned pairs in a set *audit_response* (Line 6). For each R_j, when a value is added to *audit_response*, p_i checks if there is a process that read $\perp$ from R_j (Line 7). If this is not the case, then the value in R_j is considered safe, and is added to *safe_values* (Line 8). Finally, it returns the maximum value in *safe_values* (Line 9). (We assume, for simplicity, that the input values are from a totally-ordered set.)

▶ **Lemma 4.** *A process p_i adds a value v to safe_values, only if v was proposed by some process.*

Proof. Process p_i adds values it reads from $R_0, \ldots, R_{n-1}$, the registers of other processes, to *safe_values* in Line 8. The value read from a register R_j, in Line 4, is either $\perp$ (the initial value of the register) or the value proposed by p_j, written to R_j in Line 2.

We next argue that p_i does not add $\perp$ to *safe_values*. If p_i read $\perp$ from some R_j, then since its audit on R_j follows its read from R_j, the π-Completeness of its audit operation (Definition 1(3)) implies that $(p_i, \perp)$ is contained in the response of $R_j.audit()$. By the condition of Line 7, $\perp$ is not added to *safe_values*. ◀

▶ **Lemma 5.** *Algorithm 2 satisfies validity.*

Proof. A process decides on a value in *safe_values* (Line 9), and by Lemma 4, this set contains only values proposed by some process. We complete the proof by showing that *safe_values* is not empty.

Let p_k be the first process to apply its write of v_k to R_k. Since all processes read the registers of the other processes after applying the write, it follows that all the processes read $v_k \neq \perp$ from R_k. By π-Accuracy (Definition 1(3)), the audit of R_k does not contain a pair $(p_j, \perp)$, for any p_j. Therefore, the condition in Line 7 holds and p_i adds v_k to *safe_values* in Line 8, as needed. ◀

Algorithm 2 n-process consensus using swmr atomic registers with multi-auditor atomic audit.

Shared Variables:
R_i, $i \in [0, n-1]$, swmr atomic registers with multi-auditor atomic audit; R_i is written by process p_i, initially $\perp$

Local Variables: ▷ **Pseudo code for process** p_i**,** $i \in [0, n-1]$
values[ ] an array of length n, initially $\perp$
safe_values a set, initially $\emptyset$
audit_response a set, initially $\emptyset$

```
1: propose(v_i)
2:     R_i.write(v_i)
3:     for 0 ≤ j < n
4:         values[j] ← R_j.read()
5:     for 0 ≤ j < n
6:         audit_response ← R_j.audit()
7:         if ∄(∗, ⊥) ∈ audit_response        ▷ no process read ⊥ from R_j
8:             safe_values.add(values[j])
9:     return max(safe_values)
```

▶ **Lemma 6.** *Algorithm 2 satisfies agreement.*

Proof. We prove that all processes have the same set *safe_values* when deciding, which immediately implies agreement. Suppose that process p_i is the first to add a value v_k to its *safe_values* set. This means that p_i reads v_k from register R_k (Line 4).

Let aop_i be the audit by p_i on R_k (Line 7). Since p_i adds v_k to *safe_values*, it follows that no pair $(p_j, \perp)$, for some process p_j, is included in the response to aop_i. This implies that all read operations from R_k that are linearized before aop_i do not return $\perp$.

Consider a read operation by process p_j from R_k that is linearized after aop_i. This follows the read of processes p_i from R_k, which returns $v_k \neq \perp$, and hence, this read will also read $v_k \neq \perp$. (Since only p_k writes to R_k, once, changing its value from $\perp$ to v_k.)

Thus, no read from R_k returns $\perp$. This means that any process p'_i consider $v_k \neq \perp$ in Line 7. Moreover, by the π-Accuracy property of the audit operation (Definition 1(3)), it follows that no pair $(p_j, \perp)$ is contained in the result of the audit operation by p'_i on R_k. This implies that v_k is in *safe_values* of p'_i.

Then, the *safe_values* sets of all processes are identical, and they all decide on the same value. ◀

Therefore, the algorithm satisfies validity (Lemma 5) and agreement (Lemma 6). Furthermore, all the loops in the **propose** are iterated at most n times. Since the operations invoked in the **propose** are wait-free, we get that Algorithm 2 is wait-free. This implies:

▶ **Theorem 7.** *Algorithm 2 solves consensus for n processes.*

5 Atomic audit implementations

We now turn to present several implementations of an atomic single-writer register with atomic audit. The results of the previous section indicate which synchronization primitives must be used in the implementations. Since two-process consensus can be solved with a single auditor and single reader, we cannot avoid synchronization primitives with consensus

number at least two; we use swap and fetch&add (Sections 5.1, 5.2 and 5.3). When there are multiple auditors and multiple readers, we use a universal synchronization primitive, compare&swap (Section 5.4), whose consensus number is ∞, in addition to fetch&add.

5.1 Implementing single-reader atomic register with single-auditor atomic audit using swap

We implement a swsr atomic register with single-auditor atomic audit using a swap primitive. We use a shared register R with initial value v_0, which holds the last written value, if the last operation was a write, or a special value ($\perp$) if the last operation was a read; the audit operations do not affect the value of R. In a write(v) operation, the (single) writer applies *swap* to R, atomically writing v into R and retrieving the overwritten value to check if the reader read the previously written value. In the latter case, the swap returns a special value $\perp$.

The pseudocode appears in Algorithm 3. The reader keeps the following local data structures: *val* that holds the value read from R, initially $\perp$; and *read_result* that holds the value returned by the last read operation, initially $\perp$. The writer and auditor keeps the following local data structures: *curr_val* that holds the last value written in R, initially v_0; *prev_val* that holds the previous value written into R, initially $\perp$; and a set, called *audit_result*, which stores the pairs (process,value) of the detected read operations, initially $\emptyset$.

In a read, the reader atomically reads the last value written into R and swaps it with $\perp$ to notify the writer that it read the last value written. If the response is not $\perp$, then this is the response of the read, which the reader stores in *read_result* for future read operations, before returning. Otherwise, no write has occurred since its previous read, so the read returns the value in *read_result* (without changing its value).

In a write, the writer stores in *prev_val* the value of the previous write, from *curr_val* (Line 7), and stores in *curr_val* the value v it is going to write (Line 8). In this way, if the next write operation detects that the reader has read the previous value written, the writer knows what this value is. Then, the writer swaps *curr_val* into R: atomically writing it into R and retrieving the overwritten value. If the writer gets $\perp$ from the swap, then the reader has read the last value it wrote (stored in *prev_val*), and it adds the pair (reader, *prev_val*) to *audit_result* (Line 10).

In an audit, the auditor (who is also the writer) returns all the (process,value) pairs collected during the previous write operations. By reading R, the auditor checks whether the reader read the value of the last write operation, in which case R is $\perp$. In this case, it adds the pair (reader, *curr_val*) to *audit_result*. Finally, the audit returns *audit_result*.

Fix a history H. It has at most two pending operations: one, either an audit or a write, by process p_w, and another by process p_r, which must be read. We never complete a pending audit. We complete a pending read in H if and only if some audit contains (p_r, v) in its response and no preceding read in H (which must be complete) returns v. We complete a pending write in H if and only if some read (including those completed) returns the corresponding value.

A pending operation that is completed has accessed R with a swap: A read is completed if it is the only read that returns a value detected by an audit, thus, the read has executed the swap in Line 2. A write is completed if some read has read its value, namely, the write has executed the swap in Line 9. We totally order all the completed operations by the order they apply their unique primitive on R. Call this total order π and note that it respects the real-time order of the high-level operations on the register, since the swaps and reads are inside the operations' intervals.

Algorithm 3 Implementing a single-reader atomic register with single-auditor atomic audit using swap.

Shared Variables:
R, accessed with *read* and *swap*, initially v_0

Local Variables: ▷ **Pseudo code for reader** p_r
val, initially $\perp$ ▷ result of the swap
read_result, initially $\perp$ ▷ value returned by the read

1: **Read()**
2: $val \leftarrow R.swap(\perp)$
3: **if** $val \neq \perp$
4: $read\_result \leftarrow val$
5: **return** *read_result*

Local Variables: ▷ **Pseudo code for writer and auditor** p_w
curr_val, initially v_0 ▷ last value written
prev_val, initially $\perp$ ▷ previous value written
audit_result, initially $\emptyset$ ▷ set of tuples (p, v), with p the reader and v a value

6: **Write**(v)
7: $prev\_val \leftarrow curr\_val$
8: $curr\_val \leftarrow v$
9: **if** $R.swap(v) = \perp$
10: $audit\_result.add \leftarrow (p_r, prev\_val)$
11: **return**

12: **Audit()**
13: **if** $R.read() = \perp$
14: $audit\_result.add(p_r, curr\_val)$
15: **return** *audit_result*

▶ **Lemma 8.** *Every read in π returns the value of the most recent preceding write, if there is one, and the initial value, otherwise.*

Proof. Consider a read op_r that returns a value v, and let $op_r{}'$ be the first read that returns this value. Since *read_result* is updated only if the value returned by the swap in Line 2 is not $\perp$, then the swap of $op_r{}'$ returns v. Thus, there is a preceding swap that sets R to v, and it must be by some write op_w of value v. Since reads and writes are linearized by the order of their swaps, op_w precedes $op_r{}'$, and therefore, also op_r in π.

We next argue that no other write is linearized between op_w and op_r in π. Assume otherwise, and let $op_w{}'$ be the last write that is linearized before op_r in π.

If the swap of op_r in Line 2 returns a value different from $\perp$, then this value was written by $op_w{}'$ because this is the last preceding swap that writes a non-$\perp$ value before the swap by op_r. This contradicts the assumption that op_r returns the value written by op_w.

If the swap of op_r in Line 2 returns $\perp$, this means that an earlier read that executed Line 2 after $op_w{}'$ executed its swap. The first such read swaps from R the value written by $op_w{}'$ with $\perp$. By Line 4, the value of *read_result* is not v when op_r returns in Line 5, which is a contradiction. ◀

▶ **Lemma 9.** *The set of pairs $\mathcal{P}$ returned by an audit in π satisfies the π-Completeness property.*

Proof. Consider an audit op_a that returns a set $\mathcal{P}$, and let op_r be a read returning v that precedes op_a in π. By Lemma 8, every read returns the value of the most recent preceding write in π. Let $op_r{}'$ be the first read that returns v. Then $op_r{}'$ sets R to $\perp$, and the value in *curr_val* is v.

If there is no write between op_r and op_a, the audit reads $\perp$ from R (Line 13), while *curr_val* is still v in Line 14, implying that the audit adds (p_r, v) to $\mathcal{P}$. Otherwise, there is a write between op_r and op_a. Let op_w be the first such write, and notice that op_w completes, since there is a following audit (by the same process). Moreover, since it is the first write after op_r, the value of R is $\perp$ when p_w executes Line 9 and *curr_val* is v immediately before it executes Line 7. Thus, the pair (p_r, v) is added to *audit_result*. ◀

▶ **Lemma 10.** *The set of pairs $\mathcal{P}$ returned by an audit in π satisfies the π-Accuracy property.*

Proof. Consider an audit op_a that returns a set $\mathcal{P}$, and let (p_r, v) be a pair in $\mathcal{P}$. The first operation op that adds (p_r, v) to $\mathcal{P}$ is either op_a itself or a write / audit that precedes op_a in π. This is because the variable *read_result* holding set $\mathcal{P}$ is updated immediately after the swap by p_w in the corresponding operation.

If (p_r, v) is added to $\mathcal{P}$ by an audit op, then *curr_val* is v when this happens. Since the condition in Line 13 holds, there is a reads that swaps $\perp$ into R after v was written to R. This read is between the write of v and op and by Lemma 8, it returns v, proving the lemma.

If (p_r, v) is added to $\mathcal{P}$ by a write op, then by Line 7 and Line 10, v is the value written by the write that immediately precedes op in π. Then v is the value returned by the read that swaps v with $\perp$, which allows the condition in Line 9 to hold. This read precedes op and therefore, it also precedes op_a. ◀

Lemma 8, Lemma 9 and Lemma 10 imply:

▶ **Theorem 11.** *Algorithm 3 implements a single-writer single-reader atomic register with single-auditor atomic audit.*

5.2 Implementing multi-reader atomic register with single-auditor atomic audit using swap and fetch&add

The algorithm for a *multi-reader* atomic register with single-auditor atomic audit follows a similar idea as Algorithm 3 for a *single* reader, by having each reader leave a trace of each of its reads. However, there is an additional difficulty of allowing the writer to atomically retrieve the traces of all readers when writing a new value or doing an audit.

We address this difficulty by using fetch&add, in addition to swap. A *fetch&add* allows to accurately change the value of a shared variable R so that its binary representation captures multiple pieces of information: The high-order bits hold the value written by the writer, while the n low-order bits indicate whether the readers have read the value. Specifically, the bit in position i, denoted bit_i, is associated with reader p_i, $0 \leq i < n$, and holds either 0 or 1. bit_i is set to 1 by p_i to indicate that it has read the value stored in the high-order bits of R; it is 0, otherwise. We use two functions to extract information from R. If R holds *val*, then GetValue(*val*) retrieves the value stored in the high-order bits of *val* and GetsBits(*val*) retrieves an array of n low-order bits of *val*.

In more detail (see Algorithm 4), when a reader p_i reads a value written in R, it sets bit_i to 1 by adding 2^i to the value stored in R. Since a reader can read the same value several times, p_i checks that bit_i is not already set to 1, before adding 2^i to R (Line 4). This ensures that p_i changes only its bit.

Algorithm 4 Implementation of multi-reader atomic register with single-auditor atomic audit using *fetch&add* and *swap*, for n readers.

Shared Variables:
R accessed with *read*, *swap* and *fetch&add* primitives, initially $v_0 * 2^n$

Local Variables: ▷ **Pseudo code for reader** p_i, $i \in [0, n-1]$
val, initially $\perp$ ▷ content of the register
read_result, initially $\perp$ ▷ last value read

1: **Read()**
2: $val \leftarrow R.read()$
3: **if** $(GetBits(val)[i] = 0)$
4: $read\_result \leftarrow GetValue(R.fetch\&add(2^i))$
5: **return** *read_result*

Local Variables: ▷ **Pseudo code for writer and auditor** p_w
audit_result, initially $\emptyset$ ▷ set of tuples (p, v), with p the reader and v a value
curr_val, initially v_0 ▷ last value written
prev_val, initially $\perp$ ▷ previous value written
val with initial value $\perp$ ▷ content of the register

6: **Write**(v)
7: $prev\_val \leftarrow curr\_val$
8: $curr\_val \leftarrow v$
9: $val \leftarrow R.swap(v, 0^n)$ ▷ write v in the high order bits
10: **for** $0 \leq j < n$
11: **if** $(GetBits(val)[j] = 1)$ ▷ check if p_j read the previous value
12: $audit\_result.add(p_j, prev\_val)$
13: **return**

14: **Audit()**
15: $val \leftarrow R.read()$
16: **for** $0 \leq j < n$
17: **if** $(GetBits(val)[j] = 1)$ ▷ check if p_j read the last value
18: $audit\_result.add(p_j, curr\_val)$
19: **return** *audit_result*

When writing a new value v, the writer swaps the value v and resets the n low-order bits to 0 into R and obtains the previous value of R, into a local variable *val*. Then for each reader p_i, the writer retrieves bit_i from *val* (Lines 10 and 11). If bit_i is equal to 1, the writer knows that reader p_i has read the previous value and the pair $(p_i, prev\_val)$ is added to the set to be returned by an audit, called *audit_result*. *audit_result* is a local variable, which can be accessed both by the writer and the auditor because they are the same process.

In a similar manner, an audit operation also reads R to detect high-level read operations that may have read the last value written.Since *audit_result* is a set, the pair will not be added if it was already in the set. (An efficient implementation of a sequential set can be used for this local variable.)

Fix a history H. Note that there are at most $n + 1$ pending operations in H: one (either an audit or a write) by the writer, and possibly one read operation for each reader. We never complete a pending audit. We complete a pending read invoked by process p_i in H if and

only if some audit contains (p_i, v) in its response and no earlier read in H (which must be complete) returns v to p_i. We complete a pending write in H if and only if some read in H returns the corresponding value. Note that if a pending operation is completed, then it applied a primitive to R: a write is completed if some read has read its value, namely, the write has executed the swap in Line 9; a read is completed if it is the only read that returns a value detected by an audit, thus, the read has executed the *fetch&add* in Line 4.

We totally order all the completed operations by the order they apply their last primitive (*swap*, *read* or *fetch&add*) to R. A write or an audit applies only one primitive. For a read, the last primitive is the *fetch&add*, if this is the first time that the process reads a given value, and otherwise, it is the read. Let π denote this total order, and note that it respects the real-time order of the high-level operations on the auditable register because each such step is in the execution interval of the corresponding operation. In the full version, we prove:

▶ **Theorem 12.** *Algorithm 4 implements a single-writer multi-reader atomic register with single-auditor atomic audit.*

5.3 Implementing single-reader atomic register with multi-auditor atomic audit using swap and fetch&add

The algorithm for a single-reader atomic register with *multi-auditor* atomic audit follows a similar idea as the algorithm for a *multi* reader in the previous section, using a shared register accessed with the *read*, *swap* and $fetch\&add$ primitives to support the detection of read operations by the writer and the auditors.

Since an audit operation can overlap read, write and other audit operations, we need an additional mechanism to ensure that the return value of the audit is linearizable. The reader and the auditors share information in an unbounded array of read/write registers called $pairs$, where $pairs[k]$ indicates whether the reader read the k-th value written by the writer (if there was such write). If $pairs[k]$ contains the initial value $\perp$ then the reader has not read the k-th value written, otherwise $pairs[k]$ contains that value. Each value written has a unique sequence number that is incremented when the writer performs a new write. When performing a write of a value v, the writer applies a swap to R to atomically write v together with its sequence number and set the lowest order bit of the register to 0 to indicate a new write (not yet read).

Algorithm 5 presents the pseudocode. As in Algorithm 4, in a read operation, the reader reads the value of R and sets the low-order bit to 1 if it was equal to 0 (indicating that this is the first time p_r read this value). Additionally, it writes the value read in the corresponding entry of $pairs$. When a process performs an audit operation, it retrieves from R the sequence number sn of the last write operation, and also checks whether the reader has read the last value written v. In the latter case, it writes v into $pairs[sn]$. Then, it reads all the entries of $pairs$, from index sn down to the first, to obtain its return set.

Because the value v and the sequence number are unbounded, we interleave them bit by bit in R, as done in [18]. Three functions are used to extract information from R. If R holds $val = (v, sn, bit)$, then $\mathsf{GetBit}(val)$ retrieves its lowest-order bit, $\mathsf{GetValue}(val)$ retrieves the value v, and $\mathsf{GetSn}(val)$ retrieves sn.

Note that there are at most n pending operations in H: one (either an audit or a write) by the writer, one (either an audit or a read) by the reader, and possibly one (an audit) for all the other processes. We construct a history H' by completing some operations in H. We never complete a pending audit. We complete a pending read in H if and only if some audit contains (p_r, v) in its response and no preceding read in H (which must be complete) returns v. After completing the reads, we complete a pending write if and only if some

Algorithm 5 Implementation of a single-reader atomic register with multi-auditor atomic audit using *swap* and *fetch&add*.

Shared Variables:
R accessed with *read*, *fetch&add* and *swap*. Its initial value is $(v_0, 0, 0)$.
pairs An unbounded array of read/write registers, shared by all processes. Initially all registers contains the special value $\perp$.

Local Variables: ▷ **Pseudo code for reader** p_r
val, initially $\perp$ ▷ content of the register
sn, initially 0 ▷ the sequence number of the value store in the register
read_result, initially $\perp$ ▷ value read from the register

```
1: Read()
2:     val ← R.read()
3:     if (GetBit(val) = 0)
4:         val ← GetValue(R.fetch&add(1))
5:         read_result ← GetValue(val)
6:         pairs[GetSn(val)].write(read_result)
7:     return read_result
```

Local Variables: ▷ **Pseudo code for writer** p_w
prev_val, initially $\perp$ ▷ content of the register
sn, initially 0 ▷ the sequence number of the write

```
8: Write(v)
9:     sn ← sn + 1
10:    prev_val ← R.swap((v, sn, 0))
11:    if (GetBit(prev_val) = 1)
12:        pairs[GetSn(prev_val)].write(GetValue(prev_val))  ▷ detect the read of the previous write
13:    return
```

Local Variables: ▷ **Pseudo code for auditor** p_i
audit_result, initially $\emptyset$ ▷ set of couples (process, value)
audit_index, initially 0 ▷ index of the last updated value in pairs[ ]
val, initially $\perp$ ▷ content of the register

```
14: Audit()
15:    val ← R.read()
16:    audit_index ← GetSn(val)
17:    if (GetBit(val) = 1)
18:        pairs[audit_index].write(GetValue(val))
19:    for j from audit_index to 0
20:        if(pairs[j].read() ≠⊥)
21:            audit_result.add(p_r, pairs[j].read())
22:    return audit_result
```

(completed) read returns the corresponding value. We remove from H' all other pending operations in H. Note that if a pending operation is completed, then it applied a primitive to R: a write is completed if some read has read its value, namely, the write has executed the swap in Line 10; a read is completed if it is the only read that returns a value detected by an audit, thus, the read has executed the *fetch&add* in Line 4. Thus, all operations in H' applied a primitive to R, and we can associate a sequence number *sn* to each operation, which corresponds to the sequence number they read (for a read or audit operation) or write (for a write operation) from the shared register R during this primitive.

We construct a total order π of the operations in H'. First, we put in π all the write operations according to the order they occur in H'; because write operations are executed sequentially by the unique writer, this sequence is well-defined and the order is consistent with the sequence number associated with the values written.

Next, we add the read operations in π. Since there is a unique reader the read operations are executed sequentially. The sub-sequence of read operations that returns a value with sequence number sn is placed immediately after the write operation that generates the sequence number sn, while preserving their order in H'.

The construction of π immediately implies that a read operation returns the value written by the write preceding it in π.

▶ **Lemma 13.** *Every read operation in π returns the value of the most recent preceding write in π, if there is one, and the initial value otherwise.*

Finally, we consider the audit operations one by one, in reverse order of their response in H. Consider an audit operation op_a and let sn be the sequence number it read at Line 15. There are three cases.

- Case 1: If op_a reads a value v in $pairs[sn]$, we place op_a in π immediately after the last read with sequence number sn that starts before op_a terminates.
- Case 2: If op_a read a value v in $pairs[sn-1]$ and the initial value $\perp$ in $pairs[sn]$, we place op_a in π immediately after the write operation with sequence number sn (at the start if no such write exists).
- Case 3: If op_a read the initial value $\perp$ both in $pairs[sn]$ and in $pairs[sn-1]$, then we place op_a in π immediately after the write operation with sequence number sn if it terminates before op_a is invoked in H'. Otherwise, op_a is placed immediately after the write operation with sequence number $sn-1$ (at the start if there is no such operation).

Case 3 handles the situation where an audit operation op_a reads a sequence number sn but misses a read operation op_r that returns the value with sequence number $sn-1$. This happens only if op_a is concurrent with op_r and with the write op_w that generates the sequence number sn; in particular, op_r and op_w write into $pairs[sn-1]$ after op_a read it.

In the full version, we prove that this linearization preserves the real-time order of non-overlapping operations. We next argue completeness and accuracy.

▶ **Lemma 14.** *The set of pairs $\mathcal{P}$ returned by an audit in π satisfies the π-Completeness property.*

Proof. Consider an audit operation op_a that returns a set $\mathcal{P}$, and let op_r be a read operation by process p_r that returns a value v and precedes op_a in π. We prove that $(p_r, v) \in \mathcal{P}$.

Let sn be the sequence number read by op_a, and let sn' be the sequence number of the value read by op_r. Since op_a follows op_r in π, according to our linearization rules $sn \geq sn'$. Thus, $audit\_index \geq sn'$ and op_a reads $pairs[sn']$ (Lines 19). If the read returns v, then (p_r, v) is added to $\mathcal{P}$ (Line 21), and the lemma follows. It remains to prove, by way of contradiction, that op_a does not read $\perp$ from $pairs[sn']$. We consider the possible cases:

1. $sn = sn'$, since op_a read $\perp$ from $pairs[sn']$, Case 2 or Case 3 apply. Thus, op_a is placed in π before the write that generates sequence number sn' (at the latest). Since op_r follows this write, op_a is placed before op_r in π.
2. $sn = sn' + 1$: Case 2 does not hold because op_a read $\perp$ from $pairs[sn']$ with $sn' = sn - 1$. Since op_r precedes op_a in π, by Case 3, op_a follows the write with sequence number sn in π. We are left with two cases.
   If op_a read a value $\neq \perp$ from $pairs[sn]$ (Case 1), then we reach a contradiction by showing op_a cannot read $\perp$ from $pairs[sn-1]$. Since $pairs[sn] = v'$, there is a read operation $op_{r'}$ that reads v', and since there is a single reader, $op_{r'}$ follows op_r in H. Thus, the value of $pairs[sn-1]$ is set to v before $pairs[sn]$ is set to v'. By Line 19, op_a first reads $pairs[sn]$ and then reads $pairs[sn-1]$, and therefore, it does not read $\perp$ from $pairs[sn-1]$.

Otherwise, op_a read $\perp$ from $pairs[sn]$ but the write op_w that generates sn precedes op_a (Case 3). Since op_r returns v, there is a read operation (possibly op_r) that set the corresponding low-order bit to 1 (Line 4). Then, when op_w writes the new value with sequence number sn, it reads 1 from this bit and sets $pairs[sn-1] = v$, so op_a does not read $\perp$ from $pairs[sn-1]$.

3. $sn \geq sn' + 2$. Then the write operation with sequence number $sn' + 1$ completes before op_a reads R, and we can apply the same reasoning as in case b(Case 3). ◀

▶ **Lemma 15.** *The set of pairs $\mathcal{P}$ returned by an audit in π satisfies the π-Accuracy property.*

Proof. Consider an audit operation op_a that returns a set $\mathcal{P}$, and let (p_r, v) be a pair in $\mathcal{P}$. We prove that a read operation op_r by process p_r that returns v is placed before op_a in π.

Since (p_r, v) is in $\mathcal{P}$, op_a adds (p_r, v) to *audit_result* because it read $pairs[sn] = v$ for some sn. If v was written by the reader (Line 6), then the reader returned the value v associated with sn, in Line 4. Otherwise, v was written to $pairs[sn]$ either by the writer (Line 12) or by the auditor (Line 18). This means that the writer or the auditor read the bit set to 1 when, respectively, checking the condition in Line 11 or in Line 17. This bit is set to 1 only by the reader when reading the corresponding value in Line 4.

Thus, there is a read operation op_r that read the value v with sequence number sn that does not follow op_a in H. We now show that it precedes op_a in π.

Let sn' be the sequence number of op_a. Since op_a reads $pairs[sn]$, $sn' \geq sn$. If $sn' = sn$, then since op_a read v in $pairs[sn]$, op_a is placed after op_r in π and the claim holds. If $sn' = sn + 1$, then since op_a reads v from $pairs[sn]$, by Cases 1 and 2, it is placed after the write that generates sequence number $sn+1$, and therefore, after op_r. Finally, if $sn' > sn+1$, then op_a is placed in π after a write with a sequence number greater than sn, and hence, after op_r. ◀

▶ **Theorem 16.** *Algorithm 5 implements a single-writer single-reader atomic register with multi-auditor atomic audit.*

5.4 Implementing multi-reader atomic register with multi-auditor atomic audit using compare&swap

To deal with multiple readers, as in Algorithm 4, each reader sets a dedicated bit in the n lower-order bits of a shared register R and the writer writes the value together with a sequence number in the higher-order bits of R. To deal with multiple auditors, we use an array *pairs*, as in Algorithm 5. To accommodate multiple readers, the array is bi-dimensional, with an unbounded number of columns (corresponding to each written value) and n rows, one for each reader. Specifically, $pairs[i][sn] = \perp$ indicates that process p_i has not read the value v written by the sn-th write operation (if any); otherwise, $pairs[i][sn] = v$.

We do not need readers to write into pairs because the writer applies a *compare&swap* to write the new value in R, using the previously-read state of R. So, if in the meanwhile, some reader read the current value stored in R and set its bit to 1, the *compare&swap* fails. Thus, the writer detects and write into *pairs* all the read operations of the last value written before succeeding the next write. Thus, either the auditor can detect the read operations by reading R because the bits were not reset by the new write, or this information is in *pairs*.

Because the sequence numbers and the corresponding values are unbounded, we cannot a priori divide the high-order bits between them. Instead, we interleave them bit-by-bit, as done in the previous algorithm (following [18]).

Algorithm 6 shows the pseudocode. Each process stores the value read from R in a local variable *val*, in order to select the information it needs. A read operation by a process p_i checks if low order bit associated with p_i is equal to 1 or 0, meaning p_i has already read the current value or not, respectively. We use the following functions on *val*: GetValue(*val*) returns the value stored in the high-order bits of *val*, GetSn(*val*) returns the sequence number stored in the high-order bits of *val*, and GetBits(*val*) returns the n low-order bits of *val*.

A reader has a local variable called *read_result* to store the value that has to be returned, initially $\perp$. This is used to ensure that if a new value was not written, consecutive read operations by the same process can return the correct value without setting the corresponding bit to 1 more than once.

An auditor has a local variable called *audit_result* that holds a set of pairs (process,value), one for each detected read operation; it is initially $\emptyset$. The local variable *audit_index* holds the sequence number read from R, indicating the last column of the matrix *pairs* written by the writer that the auditor has to read; it is initially 0.

In a read, the reader first reads R to check whether it has already read its last value. If this is the case, it simply returns the value. Otherwise, it applies a *fetch&add* to set its bit to 1 (indicating that it read the value) and returns the value represented by the high-order bits of the value returned by the *fetch&add*.

In an audit, the auditor first reads R to get the sequence of bits indicating the read operations performed since the last write operation and to atomically get the sequence number of the last write operation. It then reads all the pairs stored in entries of *pairs* until this index, and adds them to the set that the audit will return; this set is persistent. Finally, it adds to the set additional pairs corresponding to the low-order bits of R that are set. For simplicity, all the audit reads all entries of *pairs* starting from 0 up to the read sequence number. It is simple to ensure that the same auditor reads each column of *pairs* at most once, by using a persistent local variable to store the last column read.

To write a new value v, the writer increments the sequence number sn, and applies *compare&swap* to R to store sn together with v and reset the n low-order bits to 0. If this is successful, the operation completes; otherwise, the writer reads R, and for each of the n low-order bits that is set to 1, it writes v into the corresponding entry of $pairs[\,][sn-1]$ to announce the read operation that set the bit. The writer then retries the *compare&swap*.

We first show that all operations (by a correct process) complete within a finite number of steps. This is immediate for read operations. For an audit operation, the number of the iterations of the first *for* loop is bounded by the value of n and of *audit_index* read from R, while the second *for* loop has n iterations. The next lemma (see the full version) bounds the number of iterations in a write operation.

▶ **Lemma 17.** *The compare&swap (Line 19) of a write operation fails at most n times.*

To prove the linearizability of the algorithm, fix a history H. Note that there are at most n pending operations in H, one for each process. We construct a history H' by completing some pending read and write operations in H; we never complete a pending audit. We first complete a pending read invoked by process p_i in H if and only if some audit contains (p_i, v) in its response and no read in H (which must be complete) returns v to p_i. After completing the reads, we complete a pending write if and only if some (completed) read returns the corresponding value. We remove from H' all other pending operations in H.

Note that if a pending operation is completed, then it applied a primitive to R: A read is completed if it is the only read that returns a value detected by an audit, thus, the read has executed *fetch&add* in Line 4. A write is completed if some read has read its value, namely, the write has executed the *compare&swap* in Line 19.

Algorithm 6 Implementation of a multi-reader atomic register with multi-auditor atomic audit using *compare&swap* and *fetch&add* for n processes.

Shared Variables:
R: a register shared by all processes, accessed with *read*, *compare&swap*, and $fetch\&add$. It contains a sequence number, the corresponding value, and n bits. Initially $(0, v_0, 0^n)$.
$pairs[n, \ldots]$: a matrix of read/write registers, where $pairs[j][k]$ indicates if process p_j has read the k-th written value. Initially, $\perp$.

Local Variables: ▷ **Pseudo code for reader and auditors** p_i **(**$i \in [0, n-1]$**)**
temp initially $\perp$ ▷ the content of the register R
read_result initially $\perp$ ▷ the last value read
audit_result initially $\emptyset$ ▷ set of (process, value) pairs
audit_index initially 0 ▷ index of the last updated value in pairs[ ]

1: **Read()**
2: $temp \leftarrow R.read()$
3: **if** $(\mathsf{GetBits}(temp)[i] = 0)$
4: $read\_result \leftarrow \mathsf{GetValue}(R.fetch\&add(2^i))$
5: **return** *read_result*

6: **Audit()**
7: $temp \leftarrow R.read()$
8: $audit\_index \leftarrow \mathsf{GetSn}(temp)$
9: **for** $0 \leq j < n$
10: **for** $0 \leq k < audit\_index$
11: **if**$(pairs[j][k].read() \neq \perp)$
12: $audit\_result.add(p_j, pairs[j][k].read())$
13: **for** $0 \leq j < n$
14: **if** $(\mathsf{GetBits}(temp)[j] = 1)$ ▷ checks if p_j read the last value written in R
15: $audit\_result.add(p_j, \mathsf{GetValue}(temp))$
16: **return** *audit_result*

Local Variables: ▷ **Pseudo code for writer** p_0
temp initially $\perp$ ▷ the value read from R
sn initially 0 ▷ sequence number of the high-level writes
val initially v_0 ▷ input value of the last high-level write
bits[ ] initially 0^n ▷ n lowest-order bits of R to detect high-level reads

17: **Write(**v**)**
18: $sn \leftarrow sn + 1$
19: **while**$(R.compare\&swap((sn-1, val, bits), (sn, v, 0^n)) \neq True)$
20: $temp \leftarrow R.read()$
21: $bits \leftarrow \mathsf{GetBits}(temp)$
22: **for** $0 \leq j < n$
23: **if** $(bits[j] = 1)$ ▷ check if p_j read the last value
24: $pairs[sn-1][j].write(val)$
25: $bits \leftarrow 0^n$
26: $val \leftarrow v$
27: **return**

We construct a sequential history π that contains all the operations in H', while preserving their real-time order. We (totally) order all the read, audit and write operations in H' according to the order they apply their last primitive on R: this is either a *read* or a *fetch&add* for read operations, it is a *read* for an audit, and the *compare&swap* for write operations. Note that these are atomic primitives and their order is well-defined. Clearly, operations are linearized inside their execution intervals, implying that π preserves the real-time order of all the operations in H. In the full version, we prove that this order satisfies completeness and accuracy.

▶ **Theorem 18.** *Algorithm 6 implements a single-writer multi-reader atomic register with multi-auditor atomic audit.*

Note that all (process,value) pairs must be stored somewhere in order to allow the audit to return all the read values. When there is a single auditor, as in Section 5.1 and Section 5.2, the pairs are stored locally at the auditor. This space can be reduced if the π-Completeness property is weakened to require that the audit operation returns only the last $k \geq 1$ values read by each reader. This weaker form of completeness does not affect the consensus number.

6 The consensus number of atomic audit

The *consensus number* [12] of a concurrent object type X is the largest positive integer m such that consensus can be wait-free implemented from any number of read/write registers, and any number of objects of type X, in an asynchronous system with m processes. If there is no largest m, the consensus number is infinite.

Algorithm 1 solves consensus among two processes, using only swsr atomic registers with single-auditor atomic audit. This implies that the consensus number of a swsr atomic register with single-auditor atomic audit is at least 2. Clearly, the same holds if the register is multi-reader or multi-auditor. To prove that the consensus number of this object type is 2, it remains to prove that it is not possible to solve consensus for more than 2 processes. To this aim, we provide algorithms that implement it with a single auditor (with single or multiple readers), using a single register that supports a combination of read, swap and $fetch\&add$. In particular, Algorithm 3 implements a single-reader atomic register with a single-auditor atomic audit by applying swap and read primitive operations on a single register. Algorithm 4 implements a multi-reader atomic register with a single-auditor atomic audit by applying swap, fetch&add, and read primitives on a single register.

Herlihy [12] proves that there is no wait-free consensus algorithm for three processes using registers that support any combination of *read*, *write*, *swap* and $fetch\&add$. It follows that an atomic register with a single-auditor atomic audit cannot be used to solve consensus among three or more processes, which implies:

▶ **Proposition 19.** *A single-reader or multi-reader atomic register with a single-auditor atomic audit has consensus number two.*

Similarly, Algorithm 5 implements a single-reader atomic register with multi-auditor atomic audit. It only uses a register accessed by read, swap and fetch& add primitives, in addition to read / write registers. Herlihy's impossibility together with Theorem 3, imply:

▶ **Proposition 20.** *A single-writer single-reader atomic register with multi-auditor atomic audit has consensus number two.*

Table 1 Consensus number of atomic register with atomic audit.

Number of writers	Number of readers	Number of auditors	Consensus number
1	1	1	2
1	n	1	2
1	1	n	2
1	n	n	$\geq n$

We also show that a multi-reader atomic register with multi-auditor atomic audit has consensus number larger than 2 if each reader is also an auditor of the register. In particular, according to Algorithm 2, we can solve consensus among n processes using n-reader atomic registers with n-auditors atomic audit. Thus, the consensus number of this object type is at least n. See Table 1.

We finally provide an implementation of the swmr atomic register with multi-auditor atomic audit using read/write registers and a register accessed via read, fetch&add and compare&swap primitives. Even though an object type that supports these three primitives is not traditionally used, current architectures support this combination of primitives. Since compare&swap has infinite consensus number, it is an open question whether the consensus number of an n-reader n-auditor atomic register with atomic audit is n or more.

7 Regular Audit

A multi-reader atomic register with multi-auditor regular audit can be implemented using only single-writer multi-reader atomic registers, with a straightforward approach: during a read operation each reader leaves a trace in a register of all the values they read.

The algorithm uses several atomic registers. A swmr atomic register R_v is shared between the writer and the readers. This register is used by the writer to write a new value and by the readers to access it. In addition to R_v, each reader p_i shares a swmr atomic register $R_a[i]$ with the auditors. This register is used by p_i to communicate to the auditor all the values it read from the register.

In a read, a reader p_i reads a value from R_v and stores it in *read_result*. Then, it adds v together with its identifier in *read_log*, and writes *read_log* in the register $R_a[i]$ it shares with the auditors. Finally, it returns *read_result*. In a write, the writer simply writes v in R_v. In an audit, the auditor simply reads from all the swmr register R_a of each reader, combines it with the result in *audit_result*, and returns.

The pseudocode appears in Algorithm 7. In the full version, we prove:

▶ **Theorem 21.** *Algorithm 7 implements a single-writer multi-reader atomic register with multi-auditor regular audit.*

We remark that this algorithm can be specialized to get *single*-writer *single*-reader registers, and extended to get *multi*-writer multi-reader atomic registers. In both cases, the algorithm can provide regular audit for one or many auditors.

8 Discussion

This paper studies the synchronization power of auditing an atomic read / write register, in a shared memory system. We consider two alternative definitions of the audit operation, one that is atomic relative to the read and write operations, and another that is regular. The first

Algorithm 7 Implementation of a single-writer multi-reader atomic register with *multi-auditor regular audit* using only read and write.

Shared Variables:
R_v, swmr atomic register, initially v_0
$\forall i \in [0, n-1]$ R_a[i] swmr atomic register with writer p_i. Initially $\perp$

Local Variables: ▷ **Pseudo code for reader** p_i
read_log, initially $\emptyset$ ▷ tuples (p_j, v), for each value v read by p_j
read_result, initially $\perp$ ▷ value the reader read

1: **Read()**
2: $read\_result \leftarrow R_v.read()$
3: $read\_log.add(p_i, read\_result)$
4: $R_a[i].write(read\_log)$
5: **return** *read_result*

6: **Write**(v) ▷ Pseudo code for the writer p_0
7: $R_v.write(v)$

Local Variables: ▷ **Pseudo code for auditor** p_k
audit_result, initially $\emptyset$ ▷ tuples (p, v), with p the reader and v the value.
8: **Audit()**
9: **for** $1 \leq j < n$
10: $audit\_result.add(R_a[j].read())$
11: **return** *audit_result*

definition is shown to have a strong synchronization power, allowing to solve consensus; the number of processes that can solve consensus corresponds to the number of processes that can read and audit the register. We also implement an atomic audit operation, using swap and fetch&add for a single auditor (and multiple readers) or a single reader (and multiple auditors), and compare&swap when there are multiple readers and multiple auditors. On the other hand, the weaker, regular audit can be implemented from ordinary reads and writes.

We studied single-writer registers and leave the interesting question of registers with multiple writers to future work.

It is also interesting to investigate the precise relationship between auditable registers and DenyList objects [9], which record which processes accessed a resource and how many times. We conjecture that there are reductions between DenyList and and registers with atomic audit. The precise reductions would also explain why some variants of an auditable register with atomic audit has consensus number 2, a phenomenon that does not happen with DenyList. A reduction to DenyList, and more specifically, to the Proof-List object of [9], might also yield an implementation of an auditable register with atomic audit, and n readers and auditors, from n-consensus objects.

From a practical point of view, our results indicate that determining the precise requirements from auditable registers in real systems can be subtle, since a too-strong definition would incur high synchronization cost.

References

1 *California Consumer Privacy Act*. State of California Department of Justice `https://oag.ca.gov/privacy/ccpa`.
2 Identity Theft Resource Center. At mid-year, U.S. data breaches increase at record pace. In *ITRC*, 2018.

3 Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic. As easy as ABC: optimal (a)ccountable (b)yzantine (c)onsensus is easy! In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 – June 3, 2022*, pages 560–570. IEEE, 2022. doi:10.1109/IPDPS53621.2022.00061.

4 Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, and Adi Seredinschi. Crime and punishment in distributed byzantine decision tasks. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*, pages 34–44. IEEE, 2022. doi:10.1109/ICDCS54860.2022.00013.

5 Vinicius Vielmo Cogo and Alysson Bessani. Brief Announcement: Auditable Register Emulations. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 53:1–53:4, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.DISC.2021.53.

6 Antonella Del Pozzo, Alessia Milani, and Alexandre Rapetti. Byzantine auditable atomic register with optimal resilience. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 121–132. IEEE Computer Society, 2022. doi:10.1109/SRDS55811.2022.00020.

7 Denise Demirel, Stephan Krenn, Thomas Lorünser, and Giulia Traverso. Efficient and privacy preserving third party auditing for a distributed storage system. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 88–97. IEEE, 2016. doi:10.1109/ARES.2016.88.

8 Dipa Dharamadhikari and Sharvaree Tamne. Public auditing schemes (pas) for dynamic data in cloud: A review. In *International Conference on Smart Trends for Information Technology and Computer Communications*, pages 186–191. Springer, 2017. doi:10.1007/978-981-13-1423-0_21.

9 Davide Frey, Mathieu Gestin, and Michel Raynal. The synchronization power (consensus number) of access-control objects: The case of allowlist and denylist. In *to appear in 37th International Symposium on Distributed Computing, DISC 2023*, 2023. doi:10.4230/LIPICS.DISC.2023.21.

10 *General Data Protection Regulation*. Regulation (EU) 2016/679 https://gdpr-info.eu/.

11 Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007. doi:10.1145/1294261.1294279.

12 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

13 V Kavya, R Sumathi, and AN Shwetha. A survey on data auditing approaches to preserve privacy and data integrity in cloud computing. In *International conference on sustainable communication networks and application*, pages 108–118. Springer, 2019. doi:10.1007/978-3-030-34515-0_12.

14 Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986. doi:10.1007/BF01786228.

15 Anh Le, Athina Markopoulou, and Alexandros G Dimakis. Auditing for distributed storage systems. *IEEE/ACM Transactions on Networking*, 24(4):2182–2195, 2015. doi:10.1109/TNET.2015.2450761.

16 Bo Li, Qiang He, Feifei Chen, Hai Jin, Yang Xiang, and Yun Yang. Auditing cache data integrity in the edge computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1210–1223, 2020. doi:10.1109/TPDS.2020.3043755.

17 Jin Li, Kui Ren, and Kwangjo Kim. A2be: Accountable attribute-based encryption for abuse free access control. *Cryptology ePrint Archive*, 2009. URL: http://eprint.iacr.org/2009/118.

18 Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Recoverable and detectable Fetch&Add. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.OPODIS.2021.29.

19 *Personal Information Protection Law of the People's Republic of China*. 30th meeting of the Standing Committee of the 13th National People's Congress of the People's Republic of China on August 20.

20 Antonella Del Pozzo and Thibault Rieutord. Fork accountability in tenderbake. In Sara Tucci Piergiovanni and Natacha Crooks, editors, *5th International Symposium on Foundations and Applications of Blockchain 2022, FAB 2022, June 3, 2022, Berkeley, CA, USA*, volume 101 of *OASIcs*, pages 5:1–5:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/OASICS.FAB.2022.5.

21 Hui Tian, Yuxiang Chen, Chin-Chen Chang, Hong Jiang, Yongfeng Huang, Yonghong Chen, and Jin Liu. Dynamic-hash-table based public auditing for secure cloud storage. *IEEE Transactions on Services Computing*, 10(05):701–714, 2017. doi:10.1109/TSC.2015.2512589.

22 Boyang Wang, Baochun Li, and Hui Li. Oruta: Privacy-preserving public auditing for shared data in the cloud. *IEEE transactions on cloud computing*, 2(1):43–56, 2014. doi:10.1109/TCC.2014.2299807.

23 Jiaojiao Wu, Yanping Li, Fang Ren, and Bo Yang. Robust and auditable distributed data storage with scalability in edge computing. *Ad Hoc Networks*, 117:102494, 2021. doi:10.1016/J.ADHOC.2021.102494.

24 Yinghui Zhang, Robert H Deng, Shengmin Xu, Jianfei Sun, Qi Li, and Dong Zheng. Attribute-based encryption for cloud computing access control: A survey. *ACM Computing Surveys (CSUR)*, 53(4):1–41, 2020. doi:10.1145/3398036.

$\mathcal{O}(\log n)$-Time Uniform Circle Formation for Asynchronous Opaque Luminous Robots

Caterina Feletti
Dipartimento di Informatica, Università degli Studi di Milano, Italy

Carlo Mereghetti
Dipartimento di Informatica, Università degli Studi di Milano, Italy

Beatrice Palano
Dipartimento di Informatica, Università degli Studi di Milano, Italy

Abstract

We study the `Uniform Circle Formation` (`UCF`) problem for a distributed system of n robots which are required to displace on the vertices of a regular n-gon. We consider a well-studied model of autonomous, anonymous, mobile robots that act on the plane through Look-Compute-Move cycles. Moreover, robots are unaware of the cardinality of the system, they are punctiform, completely disoriented, opaque, and luminous. Collisions among robots are not tolerated.

In the literature, the `UCF` problem has been solved for such a model by a deterministic algorithm in the asynchronous mode, using a constant amount of light colors and $\mathcal{O}(n)$ epochs in the worst case. In this paper, we provide an improved algorithm for solving the `UCF` problem for asynchronous robots, which uses $\mathcal{O}(\log n)$ epochs still maintaining a constant amount of colors.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computing methodologies → Mobile agents; Computing methodologies → Robotic planning

Keywords and phrases Autonomous mobile robots, Opaque robots, Luminous robots, Pattern formation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.5

1 Introduction

One of the most studied classes of agent models examines systems (swarms) of autonomous and computational entities, called *robots*, which have to collaborate to solve a given problem without any central control [18, 19]. They operate through an infinite sequence of *look-compute-move* cycles, following one of the three activation and synchronization modes: the *fully synchronous* mode (FSYNCH), where time is divided in atomic *rounds* and all the robots execute a cycle synchronously in each round, the *semi-synchronous* mode (SSYNCH), which differs from the FSYNCH just for the fact that, in each round, a subset of robots executes the cycle synchronously whereas the others remain idle, and the *asynchronous* mode (ASYNCH), where there is no global clock and each robot acts asynchronously. For the SSYNCH and ASYNCH modes, time complexity is computed considering the number of *epochs* (rather than the number of rounds), where an epoch is a minimal time frame within which each robot is activated at least once.

For the sake of generality, most of the works in this field consider systems of robots with very limited features: they are *anonymous* and *indistinguishable*, they are *oblivious* (i.e. they do not have any persistent memory), they are *silent* (i.e. they cannot send messages), and they are *disoriented* (i.e. can have different local coordinate systems). In the SSYNCH and ASYNCH modes, robots do not have any information about the activation scheduling (e.g. which robots are activated or idle, or when an epoch starts). Another common feature is the robot *unawareness* of the cardinality of the system: whilst it can restrict the computational power of the system, such an unawareness condition makes the system scalable and open to

robot insertions or removals. Such minimalistic and unnatural models have been introduced in order to investigate the minimal sets of robot abilities required to solve a specific problem. At the same time, other feature assumptions are considered for exploring more realistic models: instead of the classic *punctiform* model, where robots are treated as points in space, *fat* models [2, 9, 5, 22] regard robots as solid discs. Also, instead of "more naive" models of *transparent* robots (enabling complete visibility of the system), and/or *incorporeal* robots (so that robots can occupy the same position without colliding), more recent models assume robots to be *opaque* [1, 3, 17] and acting within environments where collisions are not tolerated [15, 17, 23]. Mostly, models assume robots move in discrete spaces (grids or graphs) or in continuous spaces (typically in the Euclidean plane). Communication among agents has proven to be essential to solve some problems [4, 8, 11], especially in the ASYNCH mode where the robot motions typically must be "traffic lights coordinated". For this reason, literature has considered models of *luminous* robots, which are equipped with a light, and which can communicate and/or store some information through the colors the light can assume.

Pattern formation problems [3, 30, 31, 33] are of great importance and interest for autonomous swarms of robots; such problems require mobile entities to move on the given space and form a target pattern. The `Circle Formation` problem (requiring the swarm to achieve a *circle configuration*, i.e. a configuration where all robots lay on the same circle[1]) has been broadly investigated in the last two decades for various models [1, 9, 10, 14]. Thereafter, several algorithms have been designed to solve the "uniform" variant. The `Uniform Circle Formation` (`UCF`) problem [10, 12, 15, 17, 20, 26, 32] requires robots to arrange on the vertices of a regular n-gon, n being the cardinality of the swarm. The well-known geometric properties of a regular polygon (e.g. agreement on both origin and unit distance, equally-spaced distribution) make the `UCF` solution a fundamental algorithmic primitive for a distributed mobile system. Generally, an algorithm for `UCF` is decomposed into two steps [10, 15, 16, 17]: firstly, a `Circle Formation` solution is provided, then the algorithm solves the `Uniform Transformation` sub-problem [10] which asks robots from a circle configuration to equally distribute on the circle.

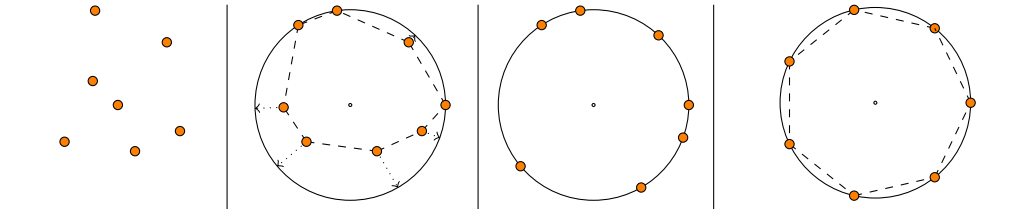
The `UCF` problem has been investigated and solved under different combinations of system features in order to point out the minimal sets of assumptions for its solution. The algorithms presented in [10] make oblivious and disoriented robots form a circle configuration and then converge toward an even distribution. The same robot model is investigated in [12] where an exact algorithm solves `UCF` for n robots, n being prime. In [25] and [26], the authors solve the problem in a swarm of fat robots, but (partially) oriented. In [20], the authors provide a solution for disoriented, and non-rigid robots, in the ASYNCH mode, avoiding collisions. In [15, 16, 17] the `UCF` problem is solved for disoriented, and opaque robots without collisions. With such strong constraints, the introduction of lights as means of communication and persistent memory turns out to be crucial. In particular, in [17] the authors provide three algorithms using $\mathcal{O}(1)$ light colors and solving the `UCF` problem in the three different synchronization modes: the ASYNCH algorithm requires $\mathcal{O}(n)$ epochs, while the synchronous algorithms solve the problem in constant time. Their FSYNCH solution improves the algorithm presented in [15] for the same model and synchronization mode. We emphasize that solving the `UCF` problem for such an opaque model but *without* lights is still an open question.

Related works and contributions. We consider the same model and the same problem as in [17]. Namely, we want to solve the `UCF` problem by opaque and luminous robots, avoiding collisions. In [17], the authors use two preliminary steps to displace the robots on the same

[1] We will always consider non-degenerate circles.

circle, upon which the regular n-gon will be formed. In particular, *(i)* they first use the algorithms in [28] (for the FSYNCH and SSYNCH modes) and [27] (for the ASYNCH mode) to displace the robots on the vertices of a convex hull, providing complete visibility to the swarm. Then, *(ii)* robots are easily made to move radially to reach their *smallest enclosing circle* (SEC). After these preliminary steps, *(iii)* authors provide original strategies to solve the `Uniform Transformation` sub-problem. Table 1 depicts the configurations obtained by executing steps *(i-iii)* in order to attain the regular polygon.

Table 1 Preliminary steps of the algorithm before reaching the Uniform Circle Configuration.



Regarding time complexity, steps *(i-ii)* need a constant number of colors and running time (rounds and epochs), in all the three synchronization modes. Concerning the original algorithms in [17] for step *(iii)*, they solve the `Uniform Transformation` sub-problem with $\mathcal{O}(1)$ colors and $\mathcal{O}(1)$ time in the FSYNCH and SSYNCH modes. On the other hand, the algorithm for step *(iii)* in the ASYNCH mode needs *linearly* many epochs in the worst case.

In this work, we present an improved solution for the `Uniform Transformation` sub-problem (i.e. step *(iii)*) in the ASYNCH mode, using a number of epochs which is *logarithmic* in the number of robots. As a corollary, this provides a $\mathcal{O}(\log n)$-time algorithm for solving the `UCF` problem from an arbitrary configuration. For the sake of conciseness, the details of steps *(i-ii)* are not repeated here: this choice allows us to focus on the novel results, without affecting the integrity and the relevance of the whole work.

Moreover, a key tool in our constructions and analysis is represented by the formalization of some geometric properties through *circular strings* [7, 21, 24, 29]. Similar approaches have been used in the realm of robot swarms [6, 12, 13]: as a matter of fact, such a formalization represents a natural and convenient tool to analyze the system configurations reached by robots. Specifically, circular strings have been used to provide robots with orientation and sorting, and to solve the *Leader Election* primitive. Moreover, specific patterns of strings have been defined to formalize particular geometric patterns: in [12], the authors solve `UCF` for asymmetric configurations by using the *Lyndon words* to elect a leader robot among the swarm. In [13], the authors introduced the *Swing words* to formalize specific configurations and solve the `UCF` problem. In this paper, we present a general discussion, providing an exhaustive and rigorous formalization of any arbitrary circle configuration through circular strings and an appropriate formal method to check the correctness of the results. As a matter of fact, several theorems and lemmas are presented (see Appendix B.2) in order to prove our strategies and outline the significant properties of robot configurations.

2 Model

We consider a system of n autonomous computational mobile robots which are *punctiform*, *anonymous*, *indistinguishable*, *homogeneous*, and *completely disoriented* (no agreement on the local coordinate system, unit distance, and chirality), operating in the Euclidean plane. They

do not know how many they are. They are *opaque*, so in the case of collinearity between three robots, the endpoint robots cannot see each other. However, they are *luminous*, i.e. they are equipped with a *persistent light* that can assume one among a constant number of colors. We say that two robots r and s *mutually see* each other if *(i)* $r = s$ or *(ii)* there is no other robot on the segment $\overline{rs}$. If r sees s, r senses only the position (in its own local coordinate system) and the color of s. Robots are given no other means to store or communicate information to the swarm. They form a distributed system where each robot executes the same *deterministic algorithm* through a sequence of *look-compute-move* cycles, in which each activated robot r takes the instantaneous snapshot (positions and lights) of the visible part of the system (*look*), executes the algorithm and computes the next position τ and light color c (*compute*), updates its light with the color c and moves straight towards τ (*move*[2]). Light color is maintained until the robot updates it in subsequent cycles. We assume that the model is *rigid*, i.e. no adversary can stop robot movement. Our robots operate in the ASYNCH mode: robots are activated independently of each other, and each *look-compute-move* cycle lasts an unpredictable but finite amount of time. Robots do not know which other robots are active at any given time. We assume the *fairness condition*: for any time t and for any robot r, there exists a time $t' > t$ such that r is activated. Our model does not tolerate either *multiplicity* (i.e. no robot can occupy the same location of another robot at the same time) or *overlapping trajectories* (robots r and s have overlapping trajectories if *(i)* r is moving from a to a', *(ii)* s is moving from b to b', and *(iii)* the segments $\overline{aa'}$ and $\overline{bb'}$ have points in common). We refer to both multiplicity and overlapping trajectories as *collisions*: hence, we aim to design algorithms that avoid collisions among robots.

3 Some notions

Given a regular n-gon, its *base angle* is the angle $\frac{2\pi}{n}$. Let $r_0, \ldots, r_{m-1}$ be m distinct robots laying ordered on the SEC according to a fixed orientation. We say that they are *consecutive* if they appear in sequence by traveling along the SEC only once. We say they are *adjacent* if, for every $0 \leq i < m-1$, no other robot lays on the arc $\overset{\frown}{r_i r_{i+1}}$.
Given a listing $r_0, \ldots, r_{n-1}$ of n adjacent robots on the SEC, centered in O, we define the sequence $\alpha_0 \cdots \alpha_{n-1}$ as the corresponding *angle-string*, where $\alpha_i = \widehat{r_i O r_{(i+1) \mod n}}$. A group of robots on an arc is *oriented* if it agrees on a common clockwise direction. Unless otherwise stated, given two points a and b on the SEC, not laying on the two different endpoints of the same diameter, we refer to the arc $\overset{\frown}{ab}$ as the *minor arc* joining a and b on the SEC.

4 Parallelism with circular strings

Once all the robots are on their SEC, we can study the geometric properties of such a disposition by considering the circular strings formed by the angle-strings in that configuration. From this viewpoint, spotting special geometric properties amounts to analyzing some properties on the circular strings.

Circular strings. Let $x = x_0 \cdots x_{n-1}$ be a *linear string* on an alphabet Σ (from now on, we simply call it *string*). We denote by $|x|$ the *length* of x, by ϵ the *empty string*, and by $x^R = x_{n-1} \cdots x_0$ the reverse string of x. A *factor* of x is ϵ or any string $x_i x_{i+1} \cdots x_j$ such that $0 \leq i \leq j \leq n-1$. We define the *k-shift* of x the string $\sigma_k(x) = x_k \cdots x_{n-1} x_0 \cdots x_{k-1}$,

[2] If τ is the current position of r, we say that r executes a *null movement*.

where $0 \leq k \leq n-1$. A string x is a *palindrome* if $x = x^R$. A *mirrored* string is a palindrome with an even length. A string x is a *power string* whenever there exists a factor y of x, such that $y \neq \epsilon$, $y \neq x$ and $x = y^k$ for an integer k. On the contrary, x is a *base string* if it is not a power string.

Given a linear string $x = x_0 \cdots x_{n-1}$, the corresponding *circular string* is the multiset $\sigma(x) = \{\sigma_0(x), \ldots, \sigma_{n-1}(x)\}$. A circular string is *minimal* if it contains only base strings. Given a string x on the alphabet Σ, we say that it is *symmetric* if *(i)* x is a palindrome, or *(ii)* $x = ayby^R$ where $y \in \Sigma^*, a, b \in \Sigma$. A circular string is said to be *symmetric* if it contains a symmetric string.

Symmetries and circular strings. Given a configuration $\mathcal{C}$ where all the robots are on the SEC and given a fixed orientation (w.l.o.g. clockwise), let us consider the listing of adjacent robots $r_0, \ldots, r_{n-1}$ on the SEC with the corresponding angle-string $\alpha = \alpha_0 \cdots \alpha_{n-1}$. From each robot r_i, two angle-strings start: $\sigma_i(\alpha)$ (clockwise) and its reverse $\sigma_{n-i}(\alpha^R)$ (counterclockwise). We denote these relations with the following notation: $r_i \curvearrowright \sigma_i(\alpha)$ and $r_i \curvearrowleft \sigma_{n-i}(\alpha^R)$. We call *configuration strings* of $\mathcal{C}$ the circular string $\sigma_{\mathcal{C}} = \sigma(\alpha)$ and $\sigma_{\mathcal{C}}^R = \sigma(\alpha^R)$. According to the properties of $\sigma_{\mathcal{C}}$ (clearly the same holds for $\sigma_{\mathcal{C}}^R$), the configuration $\mathcal{C}$ comes in three different cases of *geometric* symmetry:

- in case of asymmetry, $\sigma_{\mathcal{C}}$ is minimal and not symmetric;
- in case of symmetry with one axis, $\sigma_{\mathcal{C}}$ is symmetric. In particular, three sub-cases exist according to the number of *axis robots* (i.e. robots laying on the endpoints of the symmetry axis):
  - one axis robot (odd n), $\sigma_{\mathcal{C}}$ contains a palindrome in the form xax^R;
  - two axis robots (even n), $\sigma_{\mathcal{C}}$ contains two mirrored strings in the form xx^R and x^Rx;
  - zero axis robots (even n), $\sigma_{\mathcal{C}}$ contains two symmetric strings in the form $axbx^R$ and bx^Rax.

  In all the above three sub-cases, no other symmetric strings are contained in $\sigma_{\mathcal{C}}$;
- in case of rotational symmetry, all the strings in $\sigma_{\mathcal{C}}$ are power strings in the form x^k where x is a base string.

5 The algorithm

Let us consider a configuration $\mathcal{C}$ where all the n robots are on the same smallest enclosing circle (SEC). We explain the algorithm for the `Uniform Trasformation` sub-problem, by displacing robots on a regular n-gon inscribed on the SEC, in the ASYNCH mode. Throughout the explanation of the algorithm, we will always be using the terminology "the SEC" by meaning the *original* SEC in the configuration $\mathcal{C}$.

Regular tuple. Let $\mathcal{P}$ be the target regular n-gon, which is uniquely determined by $\mathcal{C}$ according to our algorithm. Our algorithm consists of two initial tasks where triples or 4-tuples of robots will move to some vertices of $\mathcal{P}$ and set their lights with special colors[3] (*pivot* and *angle*) to communicate "we are the reference robots". In particular, we call *regular tuple* the tuple of consecutive robots in the form *(angle, pivot, angle)* or *(angle, pivot, pivot, angle)*, laying on adjacent vertices of $\mathcal{P}$. Accordingly, it holds that, if $(r_1, \ldots, r_k)$ is a regular tuple on the SEC centered in O (with $k \in \{3, 4\}$), then $\widehat{r_i O r_{i+1}} = \frac{2\pi}{n}$ for $1 \leq i < k$.

[3] We use the *italic* notation to denote the exact color of a robot (e.g. *guard, guard_c, pivot* ...), whereas we indicate the role of a robot by the normal text (e.g. guard, pivot ...). Multiple colors can be used for the same role (e.g. a guard robot can assume the color *guard, guard_c, moving_guard* ...).

Splitting chord. Let τ, τ' be two regular tuples such that a, a' (resp. b, b') are the points where the angle robots of τ (resp. τ') lay on, such that a, a', b, b' are consecutive. Let us consider the two disjoint arcs $\overset{\frown}{ab'}$ and $\overset{\frown}{ba'}$. If in at least one of the two arcs, no regular tuple is formed (or must be formed), we say that τ and τ' are adjacent. Given two adjacent regular tuples τ, τ' defined as above, let m_a (resp. m_b) be the middle point on the minor arc aa' (resp. bb'). We call the chord $m_a\bar{m}_b$ as *splitting chord* of τ, τ'. Given a splitting chord ρ on the SEC, we call *performance arc* of ρ, denoted as $\overset{\frown}{\rho}$, the minor arc cut by ρ. We call *performance area* of ρ the region of the plane φ_ρ such that *(i)* it contains the performance arc $\overset{\frown}{\rho}$ and *(ii)* it is delimited by ρ and the two distinct straight lines, perpendicular to ρ, each one passing through one of the two endpoints of ρ. Figure 1 shows the explained elements. Note that, if ρ is a diameter, it defines two opposite performance arcs/areas.

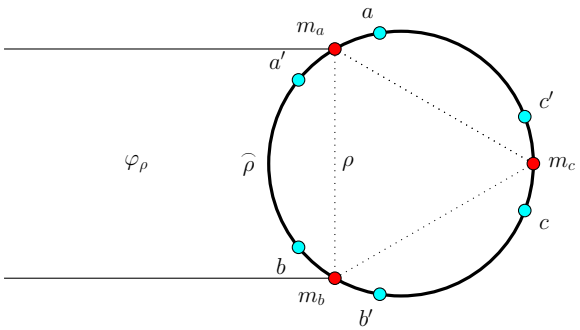


Figure 1 A splitting chord ρ, its performance arc $\overset{\frown}{\rho}$, and its performance area φ_ρ. The configuration has three splitting chords, defined by three regular triples in the form *(angle, pivot, angle)*.

Summary of the algorithm. We list the main steps and strategies used in our algorithm.

- At first, we split the SEC into performance arcs of equal length and robot cardinality, such that all the robots on each arc (or on each half-arc) are oriented. In this part, we use circular strings to formalize the geometric properties of the configurations.
- To fix the performance arcs, some robots per arc are elected and made to move in specific positions on the SEC, around each arc endpoint. These reference robots will form the regular tuples which will be used by a robot r to *(i)* reconstruct the original SEC where the regular n-gon $\mathcal{P}$ must be formed, *(ii)* recompute its base angle and so the global number of robots in the system, *(iii)* determine the performance area where r has to move, and *(iv)* detect the group of robots (on the same performance area of r) with which r has to collaborate.
- Performance areas partition the swarm into groups of robots. Each group will act independently of the others, in its own original performance area. This allows the next steps to be executed in parallel by each group.
- Within each performance area, robots equally distribute themselves on the performance arc by executing the same routine. This routine includes a loop that iterates a sequence of nine tasks, where each task is fulfilled within a constant number of epochs. Since the sequence of tasks is iterated an amount of time which is *logarithmic* in the number of robots acting in the performance area, we obtain an exponential decrease in the running time, when compared to the existing algorithms for `UCF`. Such a routine can be adapted into a stand-alone algorithm that uniformly arranges robots along a given arc.

- The time-complexity improvement from $\mathcal{O}(n)$ (see [17]) to $\mathcal{O}(\log n)$ is given by the "pairing technique" used in the task loop: robots pair themselves and use their mutual distance to encode all the needed information to reach their target destinations.

Asynchronism. The main issue with the ASYNCH mode with opaque and collision-intolerant robots is the management of moving robots which can *(i)* hide or *(ii)* be hidden by other robots. To cope with the issue *(i)*, our algorithm makes a robot color itself as *moving_x* before starting moving, where x will be the color it has to set once stopped and reactivated. A task of our algorithm is said to be *fully parallelized* if robots can safely execute their move phase (update their color or position) even if they see other *moving* robots. Instead, given a region ω on $\mathbb{R}^2$, we say that a task for a subset of robots S is ω *-synchronized* if it requires any robot $r \in S$ to skip its move phase if r sees a *moving_* colored robot in ω, different from itself. The region of synchronization of a robot can be the whole $\mathbb{R}^2$ plane or its performance area, according to the task to be executed. For the issue *(ii)*, our algorithm guarantees no collinearity with moving robots creating ambiguous snapshots. Let us show the algorithm task by task. For the sake of brevity, our explanation will omit the detail of the *moving_* colors, assuming their need and logic as understood.

5.1 SEC splitting

In the first task, some robots have to set their lights to fix the splitting chords. Moreover, at most one robot is required to travel, in a particular configuration. Starting from the configuration $\mathcal{C}$, our strategy is to select the splitting chords which split the SEC into performance arcs with the same length and with the same number of robots (possibly with just one robot of difference). The method to select such arcs depends on the geometric symmetry degree of $\mathcal{C}$.

Asymmetry. In this case, every robot can elect the lexicographically smallest string y over $\sigma_{\mathcal{C}} \cup \sigma_{\mathcal{C}}^R$ and so the robot p from which y starts (i.e. either $p \curvearrowright y$ or $p \curvearrowleft y$), such that the diameter passing through p splits the SEC into two halves with the same robot cardinality (except for just one exceeding robot at most). Note that such a diameter always exists (see Theorem 1 in [17]). In this case, the selection of y is always taken unambiguously (see Lemma 5 in Appendix B.2). The diameter passing through p is elected as the splitting chord for $\mathcal{C}$.

In this task, p sets its light as *pivot* and fixes the splitting chord. If n is even, on the other endpoint e of the splitting chord, a robot is elected to move to e where it will assume the color *pivot_a* once reactivated. For this purpose, we elect the robot already on e, if it exists, otherwise we elect the closest robot to e belonging to the most populated half-SEC.

Symmetry with just one axis. In this case, robots elect the diameter on the axis of symmetry as the splitting chord. The elected splitting chord, say d, can pass through 0, 1, or 2 opposite robots. In this task, the robots laying on d begin pivots and set their light as *pivot*. If no robot lays on d, we have to color just two robots, symmetric to d, with the color *placeholder*, to univocally fix the splitting chord (in fact, given two distinct points on a circle, there exists a unique axis of symmetry for the two points which does not pass through them). To establish an unambiguous method to elect the symmetric placeholders, we can elect the farthest robots to d (in case of equal distance, we choose according to the lexicographic order of the half-SEC angle-string).

Rotational symmetry. In this case, all strings in $\sigma(\mathcal{C})$ are power strings. Let $\alpha = \alpha_0 \cdots \alpha_{n-1}$ be an angle-string in $\sigma(\mathcal{C})$, and let β be the base string such that $\beta^k = \alpha$ (the configuration $\mathcal{C}$ can be divided into $k > 1$ identical sectors, each being the $\frac{2\pi}{k}$-rotation of the previous one). We call $\sigma(\beta)$ and $\sigma(\beta^R)$ the *sector circular strings* of the configuration. We note that they are *minimal*. Consider the ordered list of the adjacent robots on the SEC $r_0, \ldots, r_{n-1}$ which forms α. Let $P_i = \{r_j \mid j \equiv i \mod \frac{n}{k}\}$ be the *class of symmetry* which contains k robots sharing the same position in the k different sectors. We can observe that, for each robot $r \in P_i$, $r \curvearrowright \sigma_i(\alpha)$ and $r \curvearrowleft \sigma_i(\alpha)^R$ hold true. Our strategy now selects one or two classes of symmetry among $P_0, \ldots, P_{\frac{n}{k}-1}$ according to a particular property of their associated angle-strings. Observe that

▶ **Observation 1.** *Since each angle-string $\sigma_i(\alpha)$ (resp. $\sigma_i(\alpha)^R$) is the k-power of $\sigma_i(\beta)$ (resp. $\sigma_i(\beta)^R$), we can analyze the properties of the angle-strings just taking into account their factors $\sigma_i(\beta)$ and $\sigma_i(\beta)^R$.*

We say that P_i *reads* $\sigma_i(\beta)$ and $\sigma_i(\beta)^R$ (formally $P_i \curvearrowright \sigma_i(\beta)$ and $P_i \curvearrowleft \sigma_i(\beta)^R$) since the robots in P_i are the starting points for these strings (in opposite directions).

According to this, our algorithm unambiguously chooses *just one* string, say $\tilde{\gamma}$, within $\sigma(\beta) \cup \sigma(\beta^R)$, which is read exactly from *one or two* classes of symmetry. In particular, let Γ_1 and Γ_2 be the sets of strings in $\sigma(\beta) \cup \sigma(\beta^R)$, such that Γ_1 (resp. Γ_2) contains the strings read by just one class (resp. 2 classes) of symmetry. In particular, we can have two scenarios, according to the size of Γ_1:

- if $\Gamma_1 \neq \emptyset$, we unambiguously select a string $\tilde{\gamma}$ from Γ_1 (e.g. the lexicographically smallest one) and the class of symmetry, say P_i, which reads $\tilde{\gamma}$;
- if $\Gamma_1 = \emptyset$, we properly[4] select a string $\tilde{\gamma}$ from Γ_2 and the two classes of symmetry, say P_i and P_j, which read $\tilde{\gamma}$.

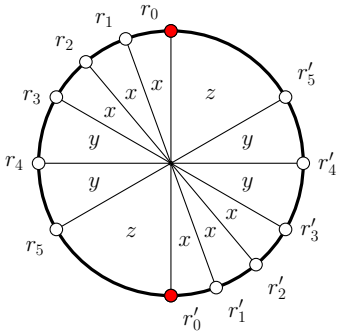
We call P_i and, if it exists, P_j *eligible classes of symmetry*. Note that we cannot have more than two classes of symmetry reading the same string in $\sigma(\beta)$ and $\sigma(\beta^R)$, otherwise the sectors would contain sub-sectors in turn, i.e. β would not be a base string (see Lemma 6 in Appendix B for the proof). Other properties about robot classes of symmetry (in particular about Γ_1, Γ_2, and so the presence of one or two eligible classes of symmetry) are explained in Appendix B.2. For example, note that, if $|\beta|$ is odd or $\sigma(\beta)$ contains a mirrored string, then Γ_1 contains at least a string, i.e. we can elect a unique eligible class (see Theorem 7 and Theorem 9 in Appendix B.2 for the proofs). Let us now show how the SEC is split in both scenarios (see Figures 2a and 2b).

One eligible class of symmetry: Each robot computes $\sigma(\beta)$ and $\sigma(\beta^R)$, and selects $\tilde{\gamma}$ and P_i as explained above. The k robots in P_i form the pivots (which won't move for the whole algorithm) by setting their color as *pivot*. The chords joining pivots belonging to adjacent sectors will be the splitting chords.

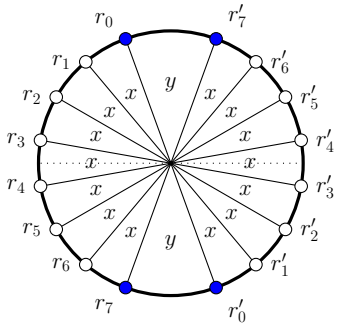
Two eligible classes of symmetry: Each robot computes $\sigma(\beta)$ and $\sigma(\beta^R)$ and selects properly $\tilde{\gamma}$, P_i, and P_j. The $2k$ robots in $P_i \cup P_j$ play the role of placeholders and set their color as *placeholder*. At the end of this task, there will be $2k$ placeholders, two for each sector. Moreover, each placeholder forms two different angle-strings with its adjacent[5] placeholders. Hence, it is important to share a common method to pair adjacent placeholders in k disjoint pairs; such pairs will be used in the next task to define the splitting chords and so delimit the k performance areas. In fact, the chords passing

[4] In order to guarantee complete visibility during regular tuple setting.

[5] Two placeholders p and r are adjacent if no other placeholder lay on the arc $\overset{\frown}{pr}$.



(a) Pivot coloring in a 2-rotation configuration with 6 classes of symmetry P_i, $i \in \{0, \dots, 5\}$. Here, $\tilde{\gamma} = x^3y^2z$ starts from just one eligible class $P_0 = \{r_0, r'_0\}$. The splitting chord is the diameter joining the two pivots.



(b) Placeholder coloring in a 2-rotation configuration with 8 classes of symmetry P_i, $i \in \{0, \dots, 7\}$. Here, $\tilde{\gamma} = x^7y$ starts from two eligible classes $P_0 = \{r_0, r'_0\}$ and $P_7 = \{r_7, r'_7\}$. The dotted diameter is elected as the splitting chord.

Figure 2 SEC splitting in the rotation case.

through the middle points of the selected pairs of placeholders will be the splitting chords. Between the two different ways to pair adjacent placeholders, robots can always choose a proper and unambiguous method to form the pairs (and so the splitting chords) so that, in the next task, the regular tuples can be formed safely. Note that for each pair of adjacent placeholders, an axis of symmetry passes between them (see Theorem 10 in Appendix B.2). Moreover, no robot lays on the endpoint of these axes of symmetry (see Theorem 11 in Appendix B.2).

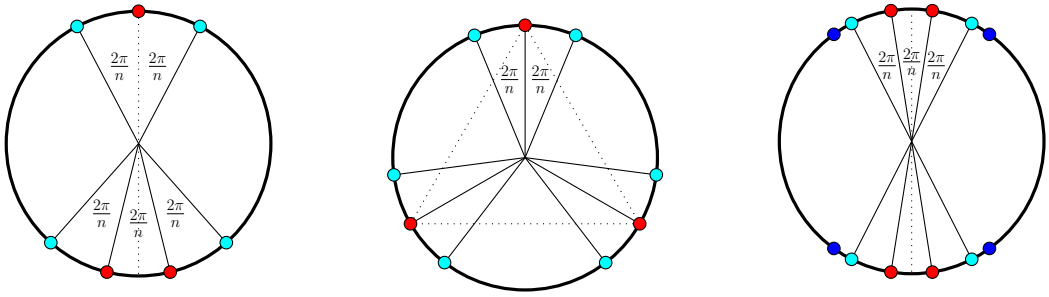
5.2 Regular tuple setting

This task requires some robots to change color and move for completing the regular tuples. The positions of these tuples are fixed by the presence of the pivots or the placeholders which have been set in the previous task. In particular, assume the current configuration presents k splitting chords $\rho_1, \dots, \rho_k$ defined by pivots and placeholders. From now on, no robot will cross the splitting chords: each robot remains in its original performance area.

Consider a splitting chord ρ, its two endpoints e_0, e_1, and its performance area φ_ρ. In this task, one or two robots per each endpoint e_i are elected on $\widehat{\rho}$, and move to complete the regular tuple around e_i in the form $(angle, pivot, angle)$ or $(angle, pivot, pivot, angle)$. The first case arises when the pivot already lays on e_i, so one robot on $\widehat{\rho}$ elects itself as an angle robot, moves in a position a on $\widehat{\rho}$ where it will set its light as *angle* for forming the base angle $\frac{2\pi}{n}$ with the pivot. Note that a second angle robot will do the same in the adjacent performance area, completing the regular triple centered on e_i. The second case arises when no robot lays on e_i, and ρ is held by *(i)* the pairs of placeholders or *(ii)* the pivot on the opposite endpoint e_{1-i} (cases of asymmetry and symmetry with odd n). In this case, two robots on $\widehat{\rho}$ elect themselves as pivot and angle, must set their light properly, and move to complete the 4-tuple centered on e_i. Figure 3 shows the regular tuple setting for different configuration types.

This task is φ_ρ -synchronized, i.e. a robot r in φ_ρ skips its move phase only if r *(i)* is not *moving_* colored and *(ii)* sees a *moving_* robot belonging to φ_ρ. In fact, other moving robots do not prevent r from executing its move phase. Note that there always exists an unambiguous strategy used by the swarm to *(i)* elect the robots which will become angle

robots and, possibly, pivots, *(ii)* make them move without colliding and always guaranteeing complete visibility to the swarm, and *(ii)* completing the k regular tuples in a constant number of epochs, potentially in parallel (see Theorem 2 in Appendix B). Eventually, the *pivot_a* robot (asymmetry case) turns its light into *pivot* aligning its color with all other pivots. Once all the regular tuples have been set, all pivots and angle robots will do nothing. Placeholders can turn off their lights since their role is no longer needed.



(a) Regular tuple setting in case of symmetry with one axis. The splitting chord lays on the axis of symmetry.

(b) Regular tuple setting in case of 3-rotation (with one eligible class). The splitting chords join the three pivots.

(c) Regular tuple setting in case of 2-rotation (with two eligible classes), around the pairs of placeholders.

Figure 3 Regular tuple setting with pivots (here red), angles (cyan), and, possibly, placeholders (blue).

5.3 Task loop

Let φ_ρ be a performance area cut by the splitting chord ρ. Let $\overset{\frown}{\rho}$ be its performance arc on the SEC, and let β_ρ be the angle-string related to the robots on $\overset{\frown}{\rho}$. If β_ρ is not a palindrome, we can establish an "upper" endpoint of $\overset{\frown}{\rho}$ according to the lexicographical orientation given by β_ρ. Otherwise, each half-arc of $\overset{\frown}{\rho}$ defines its nearest endpoint of ρ as its upper part.

Let m be the number of robots on $\overset{\frown}{\rho}$ which are not pivots or angle robots. We now show the sequence of tasks that has to be iterated until all the m robots have been moved to two particular lines external to the SEC, and parallel to ρ. As we will see, at each iteration half of the robots on $\overset{\frown}{\rho}$ leave the arc and move towards these lines, thus leading to logarithmic behavior for the whole algorithm. Note that no robot invades another performance area and the target polygon vertex of each robot is located on its performance arc.

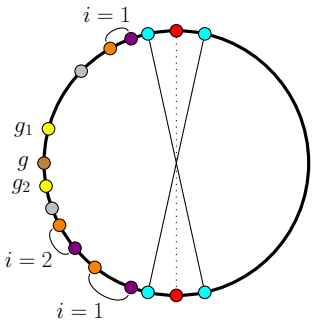
Let k be the number of performance areas on the SEC. The k groups of robots perform the same sequence of tasks independently and possibly in parallel. However, some critical tasks of the loop are φ_ρ-synchronized. Now, let us explain in detail each task of the loop (see Algorithm 1 in Appendix A for the pseudo-code of the whole task loop).

TBLG setting. In this φ_ρ -synchronized task[6], just 1 or 2 robots have to move on $\overset{\frown}{\rho}$. Let g be the middle point on $\overset{\frown}{\rho}$. We have to set 3 (if m is odd) or 2 (if m is even) guard robots around g, on the SEC, in this way:

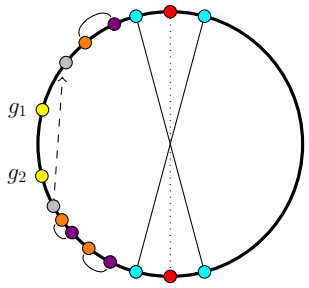
[6] The task name derives from the robot roles: top, bottom, loner, and guard.

- if m is odd, the robot closest to g (in case of two equally distant robots, the robot in the upper part of the arc is elected) moves to g and sets its color as *guard_c*[7]. We call this guard the central guard;
- the two robots closest to g (besides the one which has to head to g when m is odd) must set themselves with the *guard* color. If both are located in the same half-arc, the farthest from g must travel in the opposite half-arc, at a distance d from g which is a fraction of the minimum distance between any two robots on $\widehat{\rho}$ (this measure assures no collisions can occur). These two guards are called outer guards.

Note that no robots lay between two guards (except the central guard if m is odd). All the other robots in $\widehat{\rho}$, except for the pivots, the angles, and the guards, will color themselves according to the following roles. For each half-arc split by g, the robots pair themselves starting from the upper part of the half-arc (where the regular tuple lays): each pair is composed of the *top* robot and the *bottom* robot (in each top-bottom pair, the top robot is the closest robot to the related pivot). Each top-bottom pair must not have other robots between them. If, for example, there is an odd number of robots between the pivot and the related angle robot, the robot closest to the angle robot remains unpaired. The unpaired robots color themselves as *loner*. The final configuration is shown in Figure 4.



(a) TBLG setting with three guards. Indexing of the top-bottom pairs.



(b) TBLG disposition with two guards, and loner pairing.

Figure 4 Setting of top (here violet), bottom (here orange), loner (here lightgray), and guard robots. Top-bottom pairs are highlighted through arcs.

Loner pairing. In the previous task, at most three loners per half-arc of $\widehat{\rho}$ are created: one between the endpoint of $\widehat{\rho}$ and the pivot of the half-arc (if the pivot does not lay on ρ), one between the pivot and the angle robot, and the last one between the angle robot and its closest guard. In this task, if a half-arc presents more than one loner, the two uppermost loners (according to the upper part of each half-arc) pair themselves: the uppermost loner stays still and lights itself as *top*, while the second-uppermost one moves towards a position on the SEC in order be down with respect to the new top robot, at a relative distance from it (e.g. a fraction of the minimum distance between two robots on $\widehat{\rho}$). If just two loners remain (one per half-arc) and there is an unambiguous method to elect one of them, the elected loner will become a top robot while the second loner will approach it (using the same strategy as before) and become its bottom (see Figure 4b). This task aims to reduce the number of loners, maximizing the number of top-bottom pairs.

[7] If β_ρ is a mirrored string, then the guard is already on g.

Note that this task is φ_ρ-synchronized. At the end of this task, at most two loner robots (one per each half-arc of $\widehat{\rho}$) can exist.

Δ setting. In this φ_ρ -synchronized task, the outer guards g_1, g_2 have to approach symmetrically around g (the middle point of $\widehat{\rho}$). Let η be the minimum distance between two robots on $\widehat{\rho}$. If χ is the tangent line to the SEC passing through g, let μ be the length of the shortest projection of a segment $g\bar{g}_i$ on χ, for $i \in \{1, 2\}$. Let t be the position of the vertex of the target polygon $\mathcal{P}$ which is closest to g, but not laying on g. Let ζ be the length of the projection of the segment $\bar{gt}$ on χ. Then, each outer guard g_i moves to a position g'_i on the SEC such that g'_1 is symmetric to g'_2 with respect to g, and such that the distance between g'_1 and g'_2 is $\Delta = \min\{\eta, 2\mu, 2\zeta\}$. This measure assures *(i)* no guards turn away from g and risk colliding with other robots on the arc, and *(ii)* no guards will collide against other robots that will be moved out of the SEC in the next tasks. The measure Δ will be used as a shared value by the other robots on φ_ρ.

Pair approaching. In this fully parallelized task, every top robot r approaches its bottom robot s by traveling straight towards a point on the SEC so that the distance between r and s is δ, defined as

$$\delta = \frac{\Delta}{\langle n, m', d', i, t, c, z\rangle}$$

where

- Δ is fixed by the outer guards,
- n is the total number of robots (fixed by the base angle in the regular tuple),
- m' is the original number of robots on the arc (pivots, angles, and guards excluded),
- d' is the index of the current iteration of the loop (so, $d' = 1, \ldots, \lceil \log_2 m' \rceil$),
- i is the incremental index of the top-bottom pair starting from each endpoint of $\widehat{\rho}$ (see Figure 4a), such that the pairs closest to each endpoint have index 1 (so, $i = 1, \ldots, \lceil \frac{m'}{4} \rceil$),
- t is the number of top robots in the performance area,
- c is a flag in $\{0, 1\}$ such that if $c = 0$ then the target robot vertex is on the current half-arc where r already sits, otherwise the target vertex of r is on the other half-arc of $\widehat{\rho}$,
- z is a flag in $\{0, 1, 2\}$ such that, if $\widehat{\rho_{1/2}}$ is the half-arc of $\widehat{\rho}$ where the target vertex for r is located, then
  - $z = 0$ if the endpoints of ρ are either both covered by pivots or no pivot lays on the endpoints;
  - otherwise, $z = 1$ if the external endpoint of $\widehat{\rho_{1/2}}$ is covered by a pivot[8];
  - otherwise, $z = 2$ (i.e. if no pivot lays on the external endpoint of $\widehat{\rho_{1/2}}$).
- $\langle\ \rangle : \mathbb{N}^7 \to \mathbb{N}$ is the Cantor tuple function[9].

Note that each robot on the SEC has complete visibility of all the robots on or inside the SEC since top robots do not create collinearity with them during their movements. Note also that the setting of Δ guarantees that all top robots have to approach (and not turn away) the related bottoms to form the distance δ, avoiding collisions with other robots on

[8] The external endpoint of $\widehat{\rho_{1/2}}$ is the endpoint farthest from the guards.
[9] $\langle\ \rangle$ can be any isomorphism between $\mathbb{N}^7$ and $\mathbb{N}$.

the arc. Moreover, at the end of this task, the distance between each top and bottom robot of the same pair is always smaller than the distance between the top and the bottom robot belonging to different pairs.

Let us show how to compute d'. At each loop iteration, all the top robots are moved out of the SEC, along a specific line; namely half of the robots on the SEC leave the circle at each iteration. So each robot r on $\overset{\frown}{\rho}$ must compute the "degree of splitting" d in this way: it obtains n (from the base angle), the number k of performance areas (from the number of regular tuples), and so the number m' (where $m' < m < \frac{n}{k}$) of the original robots in its performance arc which are not pivots, angle robots or guards. Then, r counts the current q robots (pivot, angle, and guard robots excluded) in its performance arc, and computes $d = \lceil \log_2 m' \rceil - \lceil \log_2 q \rceil$, which corresponds to the index of the last iteration of the loop. Now, r knows it is starting the d'-th iteration of the loop, where $d' = d + 1$.

Top robots departure to h. Let R be the length of the radius of the SEC. Let h be the straight line parallel to the splitting chord ρ, external to the SEC, and at distance R from g (the middle point on $\overset{\frown}{\rho}$ between the outer guards). The woken top robots on $\overset{\frown}{\rho}$ can compute h, and move to this line perpendicularly, in a fully parallelized schema. Note in fact that moving robots do not obstruct other top robots from computing the next action: in fact, all the needed information to act properly is located on their performance arc.

Guard green light on. The guards can check if all the top robots have moved on h. If this is the case, the outer (resp. central) guards color themselves as *guard_green* (resp. *guard_c_green*) to tell robots on h they can proceed with the next step.

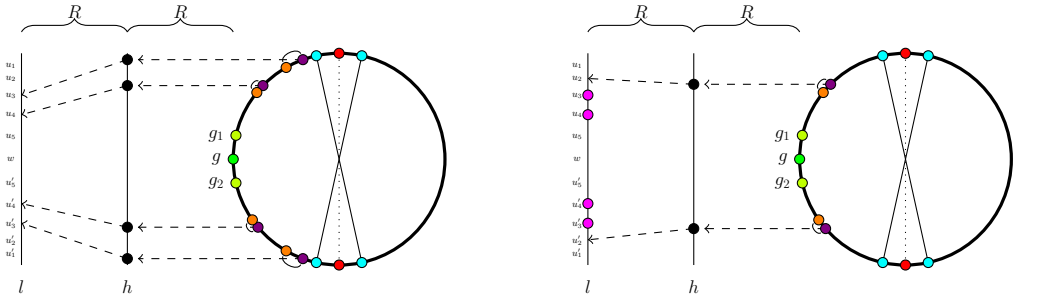
Top robots on l. Let r be a woken top robot on h. It can recompute the SEC (it can see at least two guards and at least a bottom robot), its original position on the SEC, and so the distance δ from its related bottom robot. Then, r decodes δ and recomputes the values of n, m', t, c, z, the current loop index d', and its pair index i.
Let l be the straight line parallel to h, external to the SEC, at distance R (resp. $2R$) from h (resp. the SEC). Let $\mathcal{P}$ be the target regular n-gon to be built on the SEC. Suppose m is even and let $v_1 \dots v_{\frac{m}{2}} v'_{\frac{m}{2}} \dots v'_1$ be the list of adjacent vertices of $\mathcal{P}$ on $\overset{\frown}{\rho}$ which are not already occupied by pivots and angle robots, such that they are equally distributed on the two distinct performance half-arcs. Note that the odd case is equivalent, with $v_1 \dots v_{\lfloor \frac{m}{2} \rfloor} w v'_{\lfloor \frac{m}{2} \rfloor} \dots v'_1$ as list of the vertices[10], where $w = v_{\lceil \frac{m}{2} \rceil} = v'_{\lceil \frac{m}{2} \rceil}$. Let us show how r can compute its vertex index. Thanks to the flag c, r can understand if its target vertex on $\mathcal{P}$ is on its original half-arc or on the other half-arc. This information is essential since two robots with the same index i can lay on the same half-arc: to avoid collisions, they have to know which half-arc the index refers to. Also the flag z is needed to compute the right position of the target vertex of r. Let $v_1 v_2 \dots v_{\frac{m}{2}}$ be the ordered list of the missing vertices on the target half-arc and let $u_1 u_2 \dots u_{\frac{m}{2}}$ be their projections on l. So, r chooses the vertex v_k where

$$k = \left\lfloor \frac{m}{2} \right\rfloor - \sum_{j=2}^{d'} \left\lfloor \frac{m}{2^j} \right\rfloor - \left\lceil \frac{t}{2} \right\rceil + i - 1$$

[10] As we will explain later, the vertex w is intended for the central guard.

and travels to the vertex projection u_k setting its light as *projection*. Note that this task is fully parallelized: in fact, r can compute l and δ even though all other top robots are moving toward l, since all the needed information is located "behind" r (i.e. on the SEC).

By following this schema, at each loop iteration, top robots dispose themselves on l symmetrically with respect to the central positions $u_{\frac{m}{2}}$ and $u'_{\frac{m}{2}}$, by covering the most internal free-robot projections and by leaving the two or three central positions which will be intended for the guards (see Figures 5a and 5b).



(a) Top robots travel from the SEC, to h, and then to l (after the guards have set the green light). Iteration $d' = 1$.

(b) Top robots departure from the SEC to h, and from h to l. Iteration $d' = 2$.

Figure 5 Top robots travelling from the SEC to h, and from h to l. The central positions u_5, w, u'_5 will be covered by the guards.

Gap fix. This task is executed just when on $\widehat{\rho}$ there is an odd number of bottom robots. In this case, after the previous task, there exists a projection, e.g., u_k on l where a robot lays, whereas its symmetric position u'_k is empty. From this configuration, a bottom robot can be easily elected[11] to cover the gap on l. Thus, it heads straight to u'_k and sets its color as *projection*. Note that in no other cases there exists a symmetry gap on l.

Guard green light off. The guard robots can check if no top robots still lay on h, and if, possibly, the symmetry gap on l has been covered. If so, the outer (resp. central) guard robots set their light as *guard_end* (resp. *guard_c_end*).

Loop. At this point, the algorithm re-executes the previous sequence of tasks $\mathcal{O}(\log_2 m)$ times, until all the robots on $\widehat{\rho}$ (which are not pivots, angle, or guard robots) displace themselves on the line l. Note that, in the TBLG setting, the guards do not change their positions after the first iteration, and the fixed Δ still remains the minimum distance between two robots on the same half-arc, before the *pair approaching* task.

5.4 Guards departure

At this point, the guards are the only robots on $\widehat{\rho}$ together with pivots and angle robots. Let R be the length of the radius of the SEC. Of course, the guards can compute it. In this task, each guard moves perpendicularly to l, and reaches a straight line f parallel to l and

[11] In fact, we can elect the closest robot to the guards that belongs to the half-arc with more bottom robots.

at distance R (resp. $3R$) from l (resp. from the SEC). They maintain their original colors *guard* and *guard_c* once on f. Note that their trajectories do not cross any *projection* robots on l thanks to how Δ has been chosen.

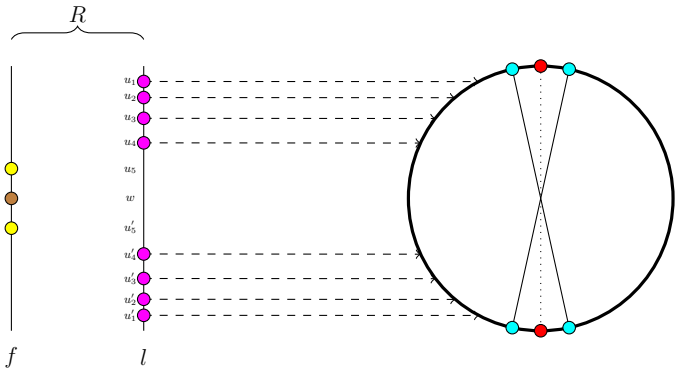


Figure 6 Robots migration from l to their target vertex on the SEC.

5.5 Back to the SEC

When all the *projection* robots on l see no more robots on the SEC (pivots and angle robots excluded), they can go back to the SEC traveling perpendicularly with respect to the line l they are placed on. Once on the SEC, they change their color in *sec* (see Figure 6).

Note that this task is fully parallelized since it is always possible for a robot r on l to reconstruct the original SEC since:

- r can see at least 3 robots on it, or
- if moving robots hide the SEC, r looks at the guards: it reconstructs f and l, and so the radius length R of the SEC and the position of its center.

5.6 Guards back to the SEC

Lastly, the guards reach the missing vertices in $\widehat{\rho}$, completing the section of the regular polygon. This task is φ_ρ -synchronized since moving guards can prevent the other guards on f to compute the needed information to act properly.

6 Conclusions

We have considered a system of n autonomous robots sharing the same features as [17]: punctiform, disoriented, indistinguishable, anonymous, opaque, and luminous. We have presented an algorithm for the `UCF` problem in the ASYNCH mode, which uses a constant number of colors and $\mathcal{O}(\log n)$ number of epochs. This work improves the previous solution in [17], which uses a linear number of epochs in the worst case. Note that the ASYNCH algorithm in [17] derives from an adaptation of a $\mathcal{O}(1)$-time algorithm for synchronized robots. Passing from synchronism to asynchronism, in fact, can require a fully-parallelizable algorithm to be executed within a one-by-one scheme, where each robot must wait until no other robot is moving before traveling. In this paper, we have designed an *ad-hoc* solution for asynchronous robots, where we have achieved a partial, but remarkable, parallelization degree. The natural investigation to pursue now concerns a possible fully-parallelizable algorithm (which uses a constant number of epochs) in the ASYNCH mode.

References

1 Ranendu Adhikary, Manash Kumar Kundu, and Buddhadeb Sau. Circle formation by asynchronous opaque robots on infinite grid. *Comput. Sci.*, 22(1), 2021. doi:10.7494/CSCI.2021.22.1.3840.

2 Kálmán Bolla, Tamás Kovács, and Gábor Fazekas. Gathering of fat robots with limited visibility and without global navigation. In *International Symposium on Swarm and Evolutionary Computation, SIDE 2012*, pages 30–38. Springer, 2012. doi:10.1007/978-3-642-29353-5_4.

3 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.

4 Kevin Buchin, Paola Flocchini, Irina Kostitsyna, Tom Peters, Nicola Santoro, and Koichi Wada. Autonomous mobile robots: Refining the computational landscape. In *35th International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021*, pages 576–585. IEEE, 2021. doi:10.1109/IPDPSW52791.2021.00091.

5 Sruti Gan Chaudhuri and Krishnendu Mukhopadhyaya. Leader election and gathering for asynchronous fat robots without common chirality. *J. Discrete Algorithms*, 33:171–192, 2015. doi:10.1016/J.JDA.2015.04.001.

6 Mark Cieliebak and Giuseppe Prencipe. Gathering autonomous mobile robots. In *9th International Colloquium on Structural Information and Communication Complexity, SIROCCO 2002*, pages 57–72. Carleton Scientific, 2002.

7 James D. Currie and D. Sean Fitzpatrick. Circular words avoiding patterns. In *6th International Conference on Developments in Language Theory, DLT 2002*, pages 319–325. Springer, 2002. doi:10.1007/3-540-45005-X_28.

8 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.

9 Suparno Datta, Ayan Dutta, Sruti Gan Chaudhuri, and Krishnendu Mukhopadhyaya. Circle formation by asynchronous transparent fat robots. In *9th International Conference on Distributed Computing and Internet Technology, ICDCIT 2013*, pages 195–207. Springer, 2013. doi:10.1007/978-3-642-36071-8_15.

10 Xavier Défago and Akihiko Konagaya. Circle formation for oblivious anonymous mobile robots with no common sense of orientation. In *Workshop on Principles of Mobile Computing, POMC 2002*, pages 97–104. ACM, 2002. doi:10.1145/584490.584509.

11 Mattia D'Emidio, Daniele Frigioni, and Alfredo Navarra. Characterizing the computational power of anonymous mobile robots. In *36th International Conference on Distributed Computing Systems, ICDCS 2016*, pages 293–302. IEEE Computer Society, 2016. doi:10.1109/ICDCS.2016.58.

12 Yoann Dieudonné and Franck Petit. Circle formation of weak robots and lyndon words. *Inf. Process. Lett.*, 101(4):156–162, 2007. doi:10.1016/J.IPL.2006.09.008.

13 Yoann Dieudonné and Franck Petit. Swing words to make circle formation quiescent. In *14th International Colloquium on Structural Information and Communication Complexity, SIROCCO 2007*, pages 166–179. Springer, 2007. doi:10.1007/978-3-540-72951-8_14.

14 Yoann Dieudonné and Franck Petit. Squaring the circle with weak mobile robots. In *19th International Symposium on Algorithms and Computation, ISAAC 2008*, pages 354–365. Springer, 2008. doi:10.1007/978-3-540-92182-0_33.

15 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 2018*, pages 317–332. Springer, 2018. doi:10.1007/978-3-030-03232-6_21.

16 Caterina Feletti, Carlo Mereghetti, and Beatrice Palano. Uniform circle formation for fully, semi-, and asynchronous opaque robots with lights. *Applied Sciences*, 13(13), 2023. doi:10.3390/app13137991.

17 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*, pages 207–221. CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3284/8511.pdf.

18 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.

19 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.

20 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.

21 László Hegedüs and Benedek Nagy. Representations of circular words. In *14th International Conference on Automata and Formal Languages, AFL 2014*, pages 261–270, 2014. doi:10.4204/EPTCS.151.18.

22 Manash Kumar Kundu, Pritam Goswami, Satakshi Ghosh, and Buddhadeb Sau. Arbitrary pattern formation by opaque fat robots on infinite grid. *Int. J. Parallel Emergent Distributed Syst.*, 37(5):542–570, 2022. doi:10.1080/17445760.2022.2088750.

23 Giuseppe Antonio Di Luna, Paola Flocchini, Sruti Gan Chaudhuri, Nicola Santoro, and Giovanni Viglietta. Robots with lights: Overcoming obstructed visibility without colliding. In *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2014*, pages 150–164. Springer, 2014. doi:10.1007/978-3-319-11764-5_11.

24 R.C. Lyndon and M.P. Schützenberger. The equation $a^M = b^N c^P$ in a free group. *Michigan Math. J.*, 9:289–298, 1962. doi:10.1307/mmj/1028998766.

25 Moumita Mondal and Sruti Gan Chaudhuri. Uniform circle formation by mobile robots. In *19th International Conference on Distributed Computing and Networking, ICDCN 2018*, pages 20:1–20:2. ACM, 2018. doi:10.1145/3170521.3170541.

26 Moumita Mondal and Sruti Gan Chaudhuri. Uniform circle formation by swarm robots under limited visibility. In *16th International Conference Distributed Computing and Internet Technology, ICDCIT 2020*, pages 420–428. Springer, 2020. doi:10.1007/978-3-030-36987-3_28.

27 Gokarna Sharma, Ramachandran Vaidyanathan, and Jerry L. Trahan. Constant-time complete visibility for asynchronous robots with lights. In *19th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2017*, pages 265–281. Springer, 2017. doi:10.1007/978-3-319-69084-1_18.

28 Gokarna Sharma, Ramachandran Vaidyanathan, Jerry L. Trahan, Costas Busch, and Suresh Rai. Complete visibility for robots with lights in O(1) time. In *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2016*, pages 327–345, 2016. doi:10.1007/978-3-319-49259-9_26.

29 Yossi Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2(2):107–121, 1981. doi:10.1016/0196-6774(81)90013-4.

30 Kazuo Sugihara and Ichiro Suzuki. Distributed algorithms for formation of geometric patterns with many mobile robots. *J. Field Robotics*, 13(3):127–139, 1996. doi:10.1002/(SICI)1097-4563(199603)13:3\%3C127::AID-ROB1\%3E3.0.CO;2-U.

31 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.

32 Giovanni Viglietta. Uniform circle formation. In *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, pages 83–108. Springer, 2019. doi:10.1007/978-3-030-11072-7_5.

33 Masafumi Yamashita and Ichiro Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theor. Comput. Sci.*, 411(26-28):2433–2453, 2010. doi:10.1016/J.TCS.2010.01.037.

A Algorithm

Algorithm 1 Pseudo-code of the *Task Loop* on the performance area φ_ρ.

Input: A performance arc $\widehat{\rho}$ where robots lay on.

$m \leftarrow$ robots on $\widehat{\rho}$ which are not pivot or angle;
while $m \geq 4$ **do**
  TBLG setting:
    The m robots color themselves as *top*, *bottom*, *loner*, *guard*, and *guard_c* in case;
    Guards arrange themselves as in Figure 4a and Figure 4b;
  Loner pairing:
    The uppermost loners move to pair themselves (see Figure 4b);
  Δ setting:
    if guards do not see any *projection* robot in φ_ρ **then**
      Outer guards approach each other to fix the distance Δ;
    end if
  Pair approaching:
    Each top robot approaches its bottom robot to form a distance δ;
  Top robots departure:
    Each top robot travels towards line h, perpendicularly;
  Guard green light on:
    Guards color themselves as *guard_green* or *guard_c_green*;
  Top robots on l:
    Top robots on h recompute the SEC and their target position on it;
    Top robots move to the projection of their target position on line l, setting their color as *projection*;
  Gap fix:
    if the number of bottom robots is odd **then**
      A bottom robot colors as *projection* and travels to l on the missing projection;
    end if
  Guard green light off:
    Guards set their color as *guard_end* or *guard_c_end*;
end while
The remaining robots on $\widehat{\rho}$, which are not pivots, angles or guards, reach l;

Output: Performance area φ_ρ where all robots lay on l (*projection* robots) or on $\widehat{\rho}$ (pivots, angles, guards).

B Lemmas and Theorems

In this section, all the lemmas and theorems used in the current paper are stated and proved.

B.1 Collision-free trajectories

▶ **Theorem 2.** *Let us assume a circle configuration where all robots lay on the SEC. Let $\widehat{a}$ be an arc of the SEC, and let $T = \{t_1, \ldots, t_h\}$ be h distinct robot-free (target) points of $\widehat{a}$, no one lying on the middle point of $\widehat{a}$. Assume at least h robots lay on distinct points of $\widehat{a} \setminus T$. Then, there exists an unambiguous way to elect h robots on $\widehat{a}$ and make them reach the h targets without colliding, and always guaranteeing complete visibility on the whole configuration.*

Proof. Firstly, we need an unambiguous way to elect h robots which will reach the target points. For this purpose, we select the h robots which are closer to the target points. In case of two robots equidistant from the same target point, we can use the geometry of the configuration (the distance from the closest endpoint of $\widehat{a}$) to elect one of them.
Let $w = w_1 \dots w_{2h}$ be the boolean string that represents the ordered displacement of the elected robots and the target points on $\widehat{a}$ such that: $w_i = 0$ symbolizes a target point, whereas $w_i = 1$ symbolizes a robot (e.g. $w = 11100010$ in Figure 7, following the clockwise direction). Let us factorize w in the shortest factors such that each factor has the same number of 0 and 1. Indeed, this factorization is unique and each factor cannot be a palindrome. Let w' be a factor of w according to the above factorization. We indicate with 1_i (resp. 0_i) the i-th 1 (resp. 0) in such a factor, where $1 \leq i \leq |\frac{w'}{2}|$. So, robots in that factor move following the following sequential schema:

for $i = |\frac{w'}{2}| \dots 1$ **do**
  if no robot in $\widehat{a}$ is moving **then**
    robot symbolized by 1_i heads to the target point symbolized by 0_i
  end if
end for

This schema assures no moving robot creates collinearity with other robots in the swarm, thus guaranteeing complete visibility. This is easily provable by induction on $|\frac{w'}{2}|$. Moreover, if h is constant, this strategy accomplishes the task in $\mathcal{O}(1)$ epochs. ◀

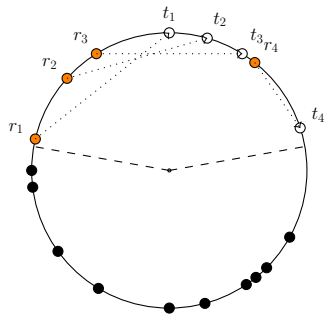


Figure 7 Trajectories of four robots assuring complete visibility to all the swarm. If a robot is moving, no other robot on the same arc moves.

B.2 Circular strings

In this section, we present some results about circular string properties which are used to study and analyze some geometric configurations of the robots on the SEC, especially in the rotational symmetry case. Specifically, given a configuration $\mathcal{C}$, we aim to figure out some properties on Γ_1, Γ_2 (i.e. on the presence of one or two eligible classes) thanks to the string properties of the sectors circular string $\sigma(\beta), \sigma(\beta^R)$ of $\mathcal{C}$.

▶ **Lemma 3** (Lyndon and Schützenberger, [24]). *Let x and y be two non-empty strings. If $xy = yx$, then there exist a string z and two integers n and m, such that $x = z^n$ and $y = z^m$.*

Proof. Let us proceed by induction on $|y| - |x|$, assuming w.l.o.g. that $|x| \leq |y|$.
If $|x| = |y|$ then the fact holds: in fact $x = y$, thus $z = x = y$ and $m = n = 1$. Otherwise, since $yx = xy$, y starts with a prefix that is equal to x. So $y = xu \Rightarrow\ xux = xxu \Rightarrow\ ux = xu$. By induction hypothesis, there exist z, m, n such that $u = z^m$, $x = z^n$, then $y = z^{n+m}$. ◀

▶ **Lemma 4.** *Let x and y be two non-empty strings. If xx^Ryy^R is a palindrome, then there exist a string z and two integers n and m, such that $x = z^n$ and $y = z^m$.*

Proof. Let us proceed by induction on $|y| - |x|$, assuming w.l.o.g. that $|x| \leq |y|$.
If $|x| = |y|$ then the fact holds: in fact $x = y$, thus $z = x = y$ and $m = n = 1$.
Let us assume that $|x| < |y|$. Hence, since xx^Ryy^R is a palindrome, y^R ends with a suffix which is equal to x^R. So

$$y^R = u^Rx^R \Rightarrow xx^Rxuu^Rx^R \text{ is a palindrome}$$

By simplifying, we have that x^Rxuu^R is a palindrome as well. By induction hypothesis, there exist z, m, n such that $u = z^m$, $x = z^n$, then $y = z^{n+m}$. ◀

▶ **Lemma 5.** *Let $\sigma(w)$ be a minimal circular string, where $w = w_0 \dots w_{n-1}$. Then $\sigma_i(w) \neq \sigma_j(w)$ for each $i \neq j$.*

Proof. The proof follows directly from Lemma 3. In fact, let us assume that there exist two different indexes such that $\sigma_i(w) = \sigma_j(w)$. Let us call them $w' = \sigma_i(w)$ and $w'' = \sigma_j(w)$, and w.l.o.g. let us assume $i < j$. Let z be the factor $w_i \dots w_{j-1}$, let x be the factor $w_j \dots w_{n-1}$, and let y be the factor $w_0 \dots w_{i-1}$, so that $w' = zxy$, and $w'' = xyz$. By Lemma 3, we can conclude that $\sigma(w)$ contains a power string, contradicting the hypothesis. ◀

▶ **Lemma 6.** *Given the two sector circular strings $\sigma(\beta), \sigma(\beta^R)$ of a configuration $\mathcal{C}$, there exist at most two classes of robots that read the same string γ. In particular, in case of two classes P_a and P_b, they reads γ in opposite directions (clockwise or counter-clockwise).*

Proof. Let us proceed by contradiction and let us suppose there is a third class P_c which reads the same string γ as P_a in the same direction. Let $a \in P_a$, $b \in P_b$, and $c \in P_c$ three robots belonging to the same sector. Let $w' = xyz$ be the string read by a and let $w'' = yzx$ be the string read by c (x is the string between a and c). Since $w' = w''$, then we can conclude that $xyz = yzx$ (for sake of simplicity, $xs = sx$). By Lemma 3, this means that w' and w'' are the power of the same factor v, which is in contrast with the initial hypothesis (a sector circular string must be minimal). ◀

▶ **Theorem 7.** *Let $\sigma(\beta), \sigma(\beta^R)$ be the two sector circular strings of a configuration $\mathcal{C}$. If $|\beta| = q$ is odd, then Γ_1 contains at least one string. So, there exists always a unique eligible class of symmetry.*

Proof. The proof follows by the fact that *(i)* the same string in $\sigma(\beta) \cup \sigma(\beta^R)$ can be read by at most 2 classes (by Lemma 6) and *(ii)*, in the worst case, $q - 1$ (even) classes read the same string two by two. So, at least one class reads a unique string. ◀

▶ **Lemma 8.** *Let $\sigma(\beta), \sigma(\beta^R)$ be the two sector circular strings of a configuration $\mathcal{C}$, where $|\beta| = q$ is even. If $\sigma(\beta)$ contains a mirrored string, then it contains exactly two different mirrored strings, whose rotation distance is $\frac{q}{2}$.*

Proof. By hypothesis, $\sigma(\beta)$ is minimal (by definition of sector circular string) and contains a string $w' = xx^R$ (and its related x^Rx). Obviously, $x^Rx = \sigma_{\frac{q}{2}}(xx^R)$. Note that $xx^R \neq x^Rx$. In fact, if $xx^R = x^Rx$, for Lemma 3 it follows that xx^R is a power string, contradicting our hypothesis ($\sigma(\beta)$ is minimal). Let us suppose that $\sigma(\beta)$ contains another pair of palindromes, say $w'' = yy^R$ (and y^Ry), at a rotation distance $k < q/2$ from the first pair. Let us assume that $x = zs$ where $|z| = k$. Thus we have that $w' = zss^Rz^R$ and $w'' = ss^Rz^Rz$. By Lemma 4, we can conclude that $w'' = v^n$ (for some not trivial v and n), again contradicting our hypothesis about $\sigma(\beta)$. ◀

▶ **Theorem 9.** *Let $\sigma(\beta), \sigma(\beta^R)$ be the two sector circular strings of a configuration $\mathcal{C}$, where $|\beta| = q$ is even. If $\sigma(\beta)$ contains a mirrored string, then Γ_1 contains at least one string. So, there exists always a unique eligible class of symmetry.*

Proof. According to Lemma 8, there exist just 2 different palindromes in $\sigma(\beta)$, say xx^R and x^Rx, which are read by two different classes of symmetry, say P_i and P_j. In particular, P_i reads xx^R in both directions, and P_j reads x^Rx in both directions. The fact that a third class P_k reads xx^R is excluded by Lemma 4. In fact, if the same palindrome string is read from two different classes, whose distance is less than $\frac{q}{2}$, then xx^R is in the form zz^Ryy^R, contradicting the hypothesis. Since $x \neq x^R$ (otherwise $\sigma(\beta)$ would not be minimal), P_i (resp. P_j) is the unique class reading xx^R (resp. x^Rx). So, Γ_1 contains at least xx^R and x^Rx. ◀

▶ **Theorem 10.** *Let $\sigma(\beta), \sigma(\beta^R)$ be the two sector circular strings of a configuration $\mathcal{C}$, where P_i and P_j are the two classes of placeholders. For each pair of adjacent placeholders $r_i \in P_i$ and $r_j \in P_j$ (no other placeholder lays on the arc $\overset{\frown}{p_ip_j}$), an axis of symmetry passes between them.*

Proof. By hypothesis, r_i and r_j read the same angle string in opposite directions. Let us suppose that the string distance between r_i and r_j is $k < |\beta|$. Thus, r_i reads $w' = x_1 \dots x_k y$ and r_j reads $w'' = x_k \dots x_1 y^R$. Since $w' = w''$, we can conclude that $x_k \dots x_1 = x_1 \dots x_k$ and $y^R = y$ (i.e. $x_1 \dots x_k$ and y are palindrome). Thus, between the arc delimited by r_i and r_j (in both directions), there is an axis of symmetry. ◀

▶ **Theorem 11.** *Let $\sigma(\beta), \sigma(\beta^R)$ be the two sector circular strings of a configuration $\mathcal{C}$, where P_i and P_j are the two eligible classes of placeholders. Then the axis of symmetry passing through two adjacent placeholders r_i, r_j does not pass across any robot on the arc $\overset{\frown}{r_ir_j}$.*

Proof. Since r_i and r_j are placeholders, they read the same angle-string w. Let us assume by contradiction that the axis passes through a robot and let xx^R be the prefix of w which is common in r_i and r_j (which is a mirrored string, since it can be read in opposite directions and its length is even). So, r_i reads the angle-string xx^Ry, whereas r_j reads xx^Ry^R. Since $|\beta|$ and $|xx^R|$ are both even (otherwise we would have just one eligible class, by Theorem 7), then $|y|$ is even too. Since $xx^Ry = xx^Ry^R$, we have that y is a mirrored string too. Let vv^R a factorization for y. Thus, $\sigma(\beta)$ turns out to contain a rotation of w which is a mirrored string (x^Rvv^Rx). By Theorem 9, since $\sigma(\beta)$ contains a mirrored string, then there is a unique eligible class of symmetry. The assumption that the axis passes through a robot must be wrong with these hypotheses. ◀

Multi-Valued Connected Consensus: A New Perspective on Crusader Agreement and Adopt-Commit

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Jennifer L. Welch
Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA

Abstract

Algorithms to solve fault-tolerant consensus in asynchronous systems often rely on primitives such as crusader agreement, adopt-commit, and graded broadcast, which provide weaker agreement properties than consensus. Although these primitives have a similar flavor, they have been defined and implemented separately in ad hoc ways. We propose a new problem called *connected consensus* that has as special cases crusader agreement, adopt-commit, and graded broadcast, and generalizes them to handle multi-valued inputs. The generalization is accomplished by relating the problem to approximate agreement on graphs.

We present three algorithms for multi-valued connected consensus in asynchronous message-passing systems, one tolerating crash failures and two tolerating malicious (unauthenticated Byzantine) failures. We extend the definition of *binding*, a desirable property recently identified as supporting binary consensus algorithms that are correct against adaptive adversaries, to the multi-valued input case and show that all our algorithms satisfy the property. Our crash-resilient algorithm has failure-resilience and time complexity that we show are optimal. When restricted to the case of binary inputs, the algorithm has improved time complexity over prior algorithms. Our two algorithms for malicious failures trade off failure resilience and time complexity. The first algorithm has time complexity that we prove is optimal but worse failure-resilience, while the second has failure-resilience that we prove is optimal but worse time complexity. When restricted to the case of binary inputs, the time complexity (as well as resilience) of the second algorithm matches that of prior algorithms.

The contributions of the paper are first, a deeper insight into the connections between primitives commonly used to solve the fundamental problem of fault-tolerant consensus, and second, implementations of these primitives that can contribute to improved consensus algorithms.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases graded broadcast, gradecast, binding, approximate agreement

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.6

Related Version *Full Version*: `https://arxiv.org/abs/2308.04646`

Funding *Hagit Attiya*: partially supported by the Israel Science Foundation (grant 22/1425).

1 Introduction

Solving consensus in the presence of faults is a fundamental problem in distributed computing, yet it is impossible to solve deterministically in purely asynchronous systems [25]. One way to address this impossibility is to augment the system model with unreliable *failure detectors* [14]. Several algorithms in this class (e.g., [11, 27]) combine a failure detector with a mechanism for detecting whether processes have reached unanimity, in the form of an *adopt-commit* protocol [37]. In such a protocol, each process starts with a binary input value and returns a

pair (v, g) where v is one of the input values and g is either 1 or 2. The process is said to pick v as its output value; furthermore, if $g = 2$, then it *commits* to v, and if $g = 1$, then it *adopts* v. In addition to the standard validity property that the output value is the input of some correct process, an adopt-commit protocol ensures that processes commit to at most one value, and if any process commits to a value, then no process adopts the other value.

Another way to address the impossibility of consensus is to use randomization and provide only probabilistic termination. Some algorithms in this class (e.g., [36]) rely on a mechanism called *crusader agreement* [19]: Roughly, if all processes start with the same value v, they must decide on this value, and otherwise, they may pick an *undecided* value, denoted $\perp$. Other algorithms in this class (e.g., [17]) rely on *graded broadcast* [23], also called *graded crusader agreement*, *graded consensus*, or just *gradecast*. Graded broadcast can be viewed as a combination of adopt-commit with crusader agreement: the decisions are either (v, g), where v is a binary value and g is either 1 or 2, or $\perp$ (also denoted $(\perp, 0)$). As in adopt-commit, the requirement is that processes *commit* to at most one value, but in addition, if any process *adopts* a value, then no process *adopts* the other value. In a sense, the $\perp$ value allows a separation between adopting one value and adopting a different value.

Crusader agreement, adopt-commit and graded broadcast have a very similar flavor, yet it is hard to tell them apart or to pinpoint how they relate to each other. (For example, some agreement protocols, e.g., [10, 30], state that they use graded broadcast while, in fact, they rely on an adopt-commit protocol, without a $\perp$ value.) The relation between these primitives becomes apparent when they are pictorially represented, as in Figure 1, with the possible decisions represented by vertices on a chain. The different "convergence" requirements are all special cases of the requirement that processes should decide on the *same or adjacent vertices* in the relevant chain.

With binary inputs, this description of the problems resembles *approximate agreement on the* $[0, 1]$ *real interval with parameter* ϵ [20]: processes start at the two extreme points of the interval, 0 or 1, and must decide on values that are at most ϵ apart from each other. Decisions must also be valid, i.e., contained in the interval of the inputs. Approximate agreement is a way to sidestep the impossibility of solving consensus in asynchronous systems and there are many algorithms for approximate agreement (e.g., [2, 20–22]).

Indeed, crusader agreement reduces to approximate agreement with $\epsilon = \frac{1}{2}$: Run approximate agreement with your input (0 or 1) to get some output y, then choose the value in $\{0, \frac{1}{2}, 1\}$ that is closest to y (taking the smaller one if there are two such values, e.g., for $y = \frac{1}{4}$). Finally, return $\perp$ if $\frac{1}{2}$ is chosen. (A similar observation is noted in [21, 29].) Likewise, adopt-commit reduces to approximate agreement with $\epsilon = \frac{1}{3}$, and graded consensus to taking $\epsilon = \frac{1}{4}$. This connection makes it clear why *binary* crusader agreement, adopt-commit and graded broadcast can be solved in an asynchronous message-passing system, in the presence of crash and malicious (unauthenticated Byzantine) failures, within a small number of communication rounds.

In some situations, agreement must be reached on a *non-binary* value, e.g., the identity of a leader [9], or the next message to deliver in totally-ordered atomic broadcast [16]. This requires handling *multi-valued* inputs, where processes can start with an input from a set V with $|V| \geq 2$. We take inspiration from *approximate agreement on graphs* [13], in which each

Figure 1 Left: crusader agreement. Center: adopt-commit. Right: graded broadcast.

process starts with a vertex of a graph as its input and must decide on a vertex such that all decisions are *within distance one of each other* and *inside the convex hull of the inputs*. When all processes start with the same vertex, this implies they must decide on this vertex.

This paper defines a new problem, which we call *connected consensus*. Connected consensus elegantly unifies seemingly-diverse problems, including crusader agreement, graded broadcast, and adopt-commit, and generalizes them to accept *multi-valued* inputs. It can be viewed as approximate agreement on a restricted class of graphs. We primarily focus on *spider graphs* [28] consisting of a central vertex attached to which are $|V|$ paths ("branches") of length R, where R is the *refinement* parameter. We also consider a variation in which the central vertex is replaced with a clique of size $|V|$ and each branch is attached to a different vertex in the clique. (See Figures 2 and 3 in Section 3.)

Recently, the definition of binary (graded) crusader agreement was extended to include a *binding* property [3]: once the first correct process terminates, there exists a value $v \in \{0, 1\}$ such that no nonfaulty process can output v in any extension. That paper demonstrates that this property facilitates the modular design of randomized consensus algorithms that tolerate an *adaptive* adversary. We refer to [3] for an excellent description of the usage, and its pitfalls, of (graded) crusader agreement, together with a shared random coin, in randomized consensus; they show how faster (graded) crusader agreement algorithms lead to faster randomized consensus algorithms. We generalize the binding property to hold for multi-valued inputs: once the first process decides, one value is "locked", so that in all possible extensions, the decisions are on the same branch of the graph. Although the generalization is natural, we point out (in Section 3) that simply applying the original definition unchanged when inputs are multi-valued does not accomplish the desired goal when connected consensus is composed with a multi-valued shared random coin [15].

With these definitions at hand, we turn to designing algorithms for multi-valued connected consensus in asynchronous message-passing systems that tolerate crash or malicious failures and satisfy binding. There is an algorithm for approximate agreement on general graphs in the presence of malicious failures [35]. However, it requires exponential local computation and does not satisfy the binding property. We are interested in special-case graphs as described above; furthermore, we focus on the cases when the refinement parameter R equals either 1 or 2, which captures the applications of interest. Thus we exploit opportunities for optimizations to obtain better algorithms.

For crash failures, we present an algorithm for $R = 1$ and $R = 2$ with the binding property; it requires $n > 2f$, where n is the total number of processes and f is the maximum number of faulty processes, which we show is optimal. Its time complexity is R and its total message complexity is $O(n^2)$. We show that the time complexity is also optimal for reasonable resiliencies. The best previous algorithms, in [3], have slightly worse time complexity: 2 for $R = 1$ (crusader agreement) and 3 for $R = 2$ (graded crusader agreement). Moreover, both of these previous algorithms are for the binary case ($|V| = 2$) only, and cannot easily handle larger input sets.

For malicious failures, we first present a simple algorithm with binding for $R = 1$ and $R = 2$, which assumes $n > 5f$. Like the crash-tolerant algorithm, its time complexity is R and its total message complexity is $O(n^2)$. We show that the time complexity is optimal for reasonable resiliencies. Both this algorithm and our crash-tolerant algorithm derive the binding property from the inputs of the processes. That is, the assignment of input values to the processes uniquely determines which non-$\perp$ value, if any, can be decided in any execution with that input assignment. The fact that the locked value for binding is determined solely by the inputs is conducive to the development of simple and efficient algorithms. However,

Table 1 Summary of connected consensus algorithms for $R = 1$ (crusader agreement) and $R = 2$ (graded broadcast) with input set V; n is the total number of processes, f is the maximum number of faulty processes. All algorithms satisfy Binding.

failure type	crash		malicious			
algorithm	Alg. 1	[3] ($\|V\| = 2$)	Alg. 2	Alg. 3	Alg. 3 + [34]	[3] ($\|V\| = 2$)
resilience	$n > 2f$	$n > 2f$	$n > 5f$	$n > 3f$	$n > 3f$	$n > 3f$
messages	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(\|V\| \cdot n^2)$	$O(n^2)$	$O(n^2)$
time ($R = 1$)	1	2	1	5	7	5
time ($R = 2$)	2	3	2	7	9	7

we show that in the presence of malicious failures, the locked value cannot be determined solely by the inputs when $n < 5f$, even if faulty processes do not equivocate and the input set is binary (see Appendix A).

Our main algorithmic contribution is a connected consensus algorithm for $R = 1$ and $R = 2$ with binding that tolerates malicious failures. Its failure resilience is $n > 3f$; a simple proof shows that this is the optimal resilience. Its time complexity is 5 for $R = 1$ and 7 for $R = 2$, and its total message complexity is $O(|V| \cdot n^2)$, where V is the set of input values. The message complexity can be reduced to $O(n^2)$, at the cost of increasing the time complexity by 2, using a reduction from [34].

Table 1 compares our algorithms with prior work. The best previous algorithms with optimal resilience are in [3] and are for the binary case only. In [3], the algorithms are evaluated in terms of communication "rounds", giving smaller numbers than our time complexity measure; we discuss the relationship between the two measures at the end of Section 5.

To summarize, our contributions are the following:

- We define the *connected consensus* problem for inputs from a set V, with a numeric *refinement* parameter R. The problem can be reduced to real-valued approximate agreement in the binary case ($|V| = \{0, 1\}$), and is equal to approximate agreement on a specific class of graphs in the multi-valued case.
- We define the *binding property* for the multi-valued case, which previously was only defined for the binary case.
- These insights lead us to design efficient message-passing algorithms for connected consensus with $R = 1$ or 2, in the presence of crash and malicious failures, for arbitrarily large input sets. The algorithms are modular in that the $R = 2$ case is obtained by appending more communication exchanges to the $R = 1$ case.
- For crash failures, our simple algorithm is optimal in resilience, time complexity (for reasonable resiliencies), and message complexity. Its time complexity improves on the best previously known algorithms, which only handle binary inputs.
- For malicious failures, we provide two algorithms that trade off resilience and time and message complexity. One algorithm has time complexity 1 or 2 (for $R = 1$ or $R = 2$), which is optimal for reasonable resiliencies, and sends $O(n^2)$ messages, but requires $n > 5f$. The other algorithm only requires $n > 3f$, but has time complexity 5 or 7 (for $R = 1$ or $R = 2$) and sends $O(|V| \cdot n^2)$ messages. This is the same performance as the algorithms in [3] which are only for the case when $|V| = 2$.

2 Model of Computation

We assume the standard asynchronous model for n processes, up to f of which can be faulty, in which processes communicate via reliable point-to-point messages. We consider two possible types of failures: *crash* failures, when a faulty process simply ceases taking steps, and *malicious* failures, when a faulty process can change state arbitrarily and send messages with arbitrary content.

In more detail, we assume a set of n processes, each modeled as a state machine. Each process has a subset of initial states, with one state corresponding to each element of V, denoting its input. The transitions of the state machine are triggered by events. There are two kinds of *events*: spontaneous wakeup of a process and receipt of a message by a process. Note that every event is a step by some process. A transition takes the current state of the process and incoming message and produces a new state of the process and a set of messages to be sent to any subset of the processes. The state set of a process contains a collection of disjoint subsets, each one modeling the fact that a particular decision has been taken; once a process enters the subset of states for a specific decision, the transition function ensures that it never leaves that subset.

A *configuration* of the system is a vector of process states, one for each process, and a set of in-transit messages. In an initial configuration, each process is in an initial state and no messages are in transit. Given a subset of at most f processes that are "faulty" with the rest being "correct", we define an *execution* as a sequence of alternating configurations and events $C_0, e_1, C_1, \ldots$ such that:

- C_0 is an initial configuration.
- The first event for each process is wakeup. A correct process experiences exactly one wakeup, a crash-faulty process experiences at most one wakeup, and a malicious-faulty process can experience any number of wakeups.
- Suppose e_i is a step by (correct)[1] process p and let s and M be the state and set of messages resulting from p's transition function applied to p's state in C_i and m, if e_i is the receipt of message m (or nothing if e_i is a wakeup event). Then the only differences between C_i and C_{i+1} are that m is no longer in transit, M is in transit, and p's state in C_{i+1} is s. If p is malicious, then s and M can be anything.
- Every message sent by a process to a correct process is eventually received and the receipt occurs after the recipient wakes up.

Since we consider all executions that satisfy the above properties, we are capturing an "adaptive adversary", which can control the inputs, choice of faulty processes, ordering of process steps, behavior of malicious processes, and the message delays, depending on anything that has happened so far in the execution.

We study worst-case complexity measures. For communication complexity, we count the maximum, over all executions, of the number of messages sent by all the (correct) processes.

We adopt the definition in [8] for time complexity in an asynchronous message-passing system. We start by defining a *timed execution* as an execution in which nondecreasing nonnegative integers ("times") are assigned to the events, with no two events by the same process having the same time. For each timed execution, we consider the prefix ending when the last correct process decides, and then scale the times so that the maximum time that elapses between the sending and receipt of any message between correct processes is 1. We

[1] Here and throughout, the restriction to correct processes is only for the case of malicious failures.

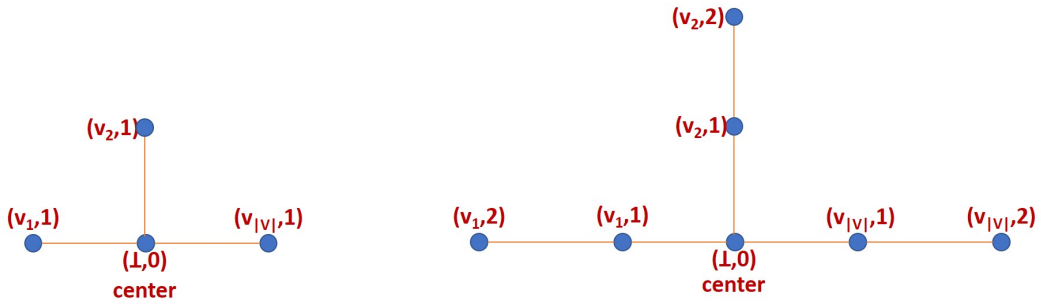


Figure 2 Spider graphs: $R = 1$ (left) and $R = 2$ (right).

define the *time complexity* as the maximum, over all such scaled timed execution prefixes, of the time assigned to the last event minus the latest time when any (correct) process wakes up. For simplicity, we assume that the first wakeup event of each process occurs at time 0. This definition of time complexity is analogous to that in [31, 34], which measures the length of the longest sequence of causally related messages.

The definition of time complexity just given applies to an algorithm with any communication structure. However, many algorithms, including several of ours, have a specific style of communication, in which each process repeatedly sends a message to all the processes, waits to receive a certain number of messages, and then sends another message. For algorithms with this structure, the sending of a message and waiting for the receipt of the specified number of messages, forms a *round*.

3 Definitions of Connected Consensus and Related Problems

Connected Consensus. Let V be a finite, totally-ordered set of values; assume $\perp \notin V$. Given a positive integer R, let $G_S(V, R)$ be the spider graph consisting of a central vertex labeled $(\perp, 0)$ that has $|V|$ paths extending from it, with one path ("branch") associated with each $v \in V$. The path for each v has R vertices on it, not counting $(\perp, 0)$, labeled $(v, 1)$ through (v, R), with (v, R) being the leaf. (See Figure 2.) Given a subset V' of V, we denote by $T(V, R, V')$ the minimal subtree of $G_S(V, R)$ that connects the set of leaves $\{(v, R)|v \in V'\}$; note that when V' is a singleton set $\{v\}$ then $T(V, R, \{v\})$ is the single (leaf) vertex (v, R).

In the *connected consensus problem for V and R*, each process has an input from V. The requirements are:

Termination: Each correct process must decide on a vertex of $G_S(V, R)$, namely, an element of $\{(v, r)|v \in V, 1 \leq r \leq R\} \cup \{(\perp, 0)\}$.

Validity: Let $I = \{(v, R)|v$ is the input of a (correct) process$\}$. The output of each (correct) process must be a vertex in $T(V, R, I)$. In particular, if all (correct) processes start with the same input v, then (v, R) must be decided.

Agreement: The distance between the vertices labeled by the decisions of all (correct) processes is at most one.

If we set $R = 1$, we get *crusader agreement* [19], originally considered in the synchronous model. If we set $R = 2$ we get *graded broadcast* [24], originally considered in the synchronous model. In asynchronous shared-memory systems, graded broadcast is also called *adopt-commit-abort* [18, 32].

Defining the Binding Property. An additional condition of interest for the connected consensus problem is called *binding* [3]. It was originally proposed for the case of binary inputs, and defined as follows: "before the first non-faulty party terminates, there is a value $v \in \{0, 1\}$ such that no non-faulty party can output the value v in any continuation of the execution." Here we generalize this property for multi-valued inputs.

Binding: In every execution prefix that ends with the first (correct) process deciding, one value is "locked", meaning that in every extension of the execution prefix, the decision of every (correct) process must be on the same branch of the spider graph.

If the first decision is not $(\perp, 0)$, then this condition follows from Agreement. More interestingly, if the first decision is $(\perp, 0)$, then there are many choices as to which branch is locked but the choice must be the same in every extension. Note that when $|V| = 2$, our definition is equivalent to the original from [3], but for larger V, our definition is stronger – the original definition only excludes one value, leaving $|V| - 1$ possible decision values, while ours excludes $|V| - 1$ values, leaving only one possible decision value.

The original definition of binding is not strong enough to handle multi-valued inputs in some cases. For example, consider the framework for solving asynchronous Byzantine Agreement by alternating calls to a (black box) connected consensus with $R = 1$ (crusader agreement) subroutine with calls to a shared random coin subroutine (cf. [3, Section 3] and [7, Section 2]). Suppose there are three processes with inputs 0, 1 and 2, and the first process, say p, to return from connected consensus gets $(\perp, 0)$ (corresponding to $\perp$ in crusader agreement). According to the original definition, some value $v \in \{0, 1, 2\}$ is no longer a possible output of the connected consensus subroutine. Since p obtains $(\perp, 0)$ from crusader agreement, it calls the shared coin; let $v' \in \{0, 1, 2\}$ be the value p gets from the shared coin. However, the original binding property still allows the adversary to make the other processes return $(v'', 1)$, with $v'' \neq v'$. This would mean that processes start the next iteration in disagreement.

When $R = 1$, there are only two vertices on any given branch of the spider graph, $(v, 1)$ and $(\perp, 0)$. This implies:

▶ **Proposition 1.** *If $R = 1$, then the Binding property implies the Agreement property.*

If $R = 2$, though, the Binding property only restricts the branch of the spider graph on which decisions can be made; both $(\perp, 0)$ and $(v, 2)$ are on the same branch, but Agreement does not permit them to both be decided.

Centerless Variants. Recall that in the *adopt-commit* problem [27, 37], processes return a pair (v, g) where v is one of the input values and g is either 1 (adopt) or 2 (commit). Thus, there is no analog of the "center" vertex. We model this with a *centerless* variant of a spider graph (see left side of Figure 3). Here, $G_S(V, R)$ is the graph consisting of a clique on the vertices $(v, 1)$ for all $v \in V$, each with a path extending from it, with $R - 1$ vertices on it, not counting $(\perp, 0)$, labeled $(v, 2)$ through (v, R), with (v, R) being the leaf. Decisions must satisfy termination, validity and agreement as specified for the variant with a center. Since the graph has no center, binding cannot be defined; indeed, when a process returns $(v, 1)$, other processes might return $(v', 1)$, for $v \neq v'$.

Instead of developing algorithms directly for the centerless problem, we note that it can be reduced to the centered problem with the same refinement parameter: Call the algorithm for the centered problem with your input u. If the return value is (v, g) with $g > 0$, then decide this value for the centerless problem; when the return value is $(\perp, 0)$ *(i.e, the center)*, decide $(u, 1)$ for the centerless problem. (See right side of Figure 3). This reduction implies the following proposition:

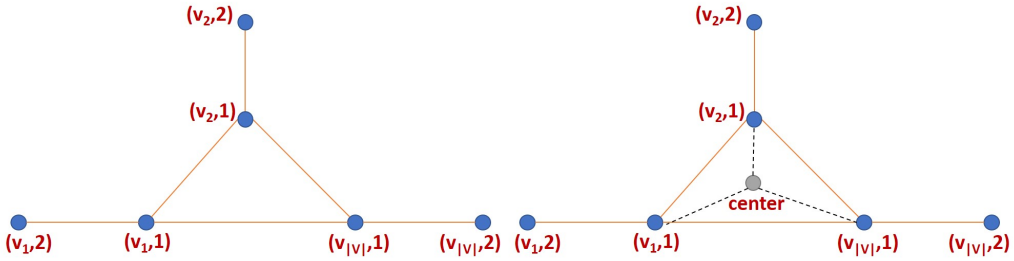


Figure 3 Centerless graph with $R = 2$ (left) and its reduction to a (centered) spider graph with $R = 2$ (right).

▶ **Proposition 2.** *If A is an algorithm that solves the (centered) connected consensus problem for R, then there is an algorithm A' that solves the centerless connected consensus problem for R.*

In the *vacillate-adopt-commit* (VAC) problem [4], the possible output values are (v,commit), (v,adopt), and (v,vacillate), where v is any value. If any output is (v,commit), then every other output is either (v,commit) or (v,adopt), for the same v. Furthermore, if there is no commit output and there is at least one (v,adopt) output, then every other output is either (v,adopt), with the same value v, or (w,vacillate), where w can be any value. At first glance, VAC seems to correspond to a centerless graph with refinement parameter $R = 3$. However, a closer look at the usage of VAC suggests that the return value of vacillate is irrelevant, in which case the problem could be represented as a centered spider graph with $R = 2$, like adopt-commit-abort and graded crusader agreement.

4 Tolerating Crash Failures

4.1 Lower Bounds

Resiliency. We first note that a standard partitioning argument shows that $n > 2f$ is required to solve connected consensus with crash failures for any $R \geq 1$, even without binding. Assume in contradiction that there is an algorithm for $n = 2f$ where $V = \{0, 1\}$ and consider execution α_0 in which all processes have input 0 and half of the processes crash initially; by Validity, the other half must decide $(0, R)$ by some time t_0. Consider execution α_1 in which all processes have input 1 and the other half of the processes crash initially; by Validity, the first half must decide $(1, R)$ by some time t_1. Finally consider execution α which is the "merger" of α_0 and α_1, in which the first half of the processes have input 0, the other half have input 1, and messages between the halves are delayed until after $\max\{t_0, t_1\}$. Since processes in the first half decide $(1, R)$ as they do in α_1 and processes in the second half decide $(0, R)$ as they do in α_0, Agreement is violated.

Round Complexity. First we note that processes cannot solve connected consensus without communicating, and thus at least one round is necessary.

For $R = 2$, we use a reduction from approximate agreement and a result in [22] to show that, as long as $n \leq 4f$, at least two rounds of message exchange are necessary to solve connected consensus.

Suppose we want to solve ϵ-approximate agreement on the interval $[0, 1]$. We show how to do so using any connected consensus algorithm A_{CC} with $V = \{0, 1\}$ and $R = \lceil \frac{1}{2\epsilon} \rceil$ that tolerates crash failures. Letting v be the approximate agreement input for process p, call A_{CC}

with input v. To obtain the approximate agreement output from the connected consensus output, map the $2R+1$ vertices of the connected consensus graph, which is a chain, in order to points in the real interval $[0,1]$ that are equally spaced, with vertex $(0,R)$ corresponding to point 0 and vertex $(1,R)$ corresponding to point 1. Since adjacent points in $[0,1]$ are distance $\frac{1}{2R}$ apart, they are within ϵ of each other.

For example, when $\epsilon = \frac{1}{4}$, we use connected consensus with $R = 2$.

This transformation uses no rounds other than those used by A_{CC}. Thus any lower bound on the number of rounds for approximate agreement is also a lower bound on the number of rounds for connected consensus.

For any round-based approximate agreement algorithm, the *convergence ratio* is the fraction by which the size of the interval of values reduces after one round. The number of rounds needed to achieve outputs that are at most ϵ apart is $\left\lceil \log_{\frac{1}{r}} \frac{U}{\epsilon} \right\rceil$, where r is the convergence ratio per round and U is the size of the interval of inputs. In our case, $U = 1$ and $\epsilon = 1/4$. Fekete proved in [22] that the best r can be is $\left\lceil \frac{n-f}{f} \right\rceil^{-1}$. Thus, as long as $n \leq 4f$, the number of rounds is at least 2.

In the full version we show that, if the resilience is higher than $4f$, a simple one-round connected consensus algorithm with Binding is possible for $R = 2$.

4.2 Algorithm

We next present an algorithm for connected consensus with binding for n processes that tolerates $f < n/2$ crash failures. (See Algorithm 1.) The algorithm is extremely simple and efficient, using one round of message exchanges for $R = 1$ and two rounds of message exchanges for $R = 2$, and thus using only $O(n^2)$ messages. Due to the simple communication structure, the number of rounds of message exchanges is equal to the time complexity. The lower bounds discussed in the previous subsection show that the time complexity is optimal for $R = 1$, and that for $R = 2$, it is optimal as long $n \leq 4f$.

In the first round of the algorithm, processes exchange their inputs. After hearing from $n - f$ processes, each process chooses branch v of the spider graph, if all the received values equal v, or the center vertex otherwise. If $R = 1$, then the process decides on the branch. Otherwise, processes exchange branch values (v or $\perp$) in a second round in order to decide on a vertex on the v branch. After waiting for $n - f$ messages in the second round, if a process' branch is $\perp$, then it decides $(v, 1)$ if at least one second-round message is for v and $(\perp, 0)$ otherwise. If the process' branch is v, then it decides $(v, 2)$ if all the second-round messages are for v and $(v, 1)$ otherwise.

▶ **Theorem 3.** *If $n > 2f$, then Algorithm 1 solves binding connected consensus for $R = 1$ and $R = 2$ with n processes, up to f of which can crash. It takes 1 time unit and sends $O(n^2)$ messages for $R = 1$ and takes 2 time units and sends $O(n^2)$ messages for $R = 2$.*

The proof of this theorem appears in the full version. Here, we only outline why the algorithm is binding. In fact, we show a stronger property, that the branch along which decisions are made is determined solely by the inputs.

For any assignment of inputs to the processes, since $n > 2f$, there is at most one input value $v \in V$ that occurs at least $n - f$ times. Note that if p sets its branch variable to v, then all $n - f$ INPUT messages it receives are for v, and since processes fail only by crashing, the $n - f$ senders of these messages all have input v. Therefore, no process can set its branch variable to any value in V other than v. For $R = 1$, processes decide on their branch variables, implying that in any future extension, every process that sets its branch variable sets it to v

Algorithm 1 Connected Consensus algorithm with Binding for $R = 1, 2$ with n processes, $f < n/2$ of which may crash; code for process p.

```
1:  send ⟨INPUT,input⟩ to all                                   ▷ round 1
2:  wait for n − f INPUT messages
3:  let W be set of values received in INPUT messages
4:  if ∃v ∈ V s.t. W = {v} then
5:      branch := v
6:  else
7:      branch := ⊥
8:  if R = 1 then
9:      if branch = ⊥ then
10:         decide (⊥, 0)                                       ▷ center vertex
11:     else
12:         decide (branch,1)                                   ▷ leaf vertex for chosen branch
13: else                                                        ▷ R = 2; round 2
14:     send ⟨BRANCH,branch⟩ to all
15:     wait for n − f BRANCH messages
16:     if branch = ⊥ then
17:         if ∃v ∈ V s.t. at least one BRANCH message has value v then
18:             decide (v, 1)                                   ▷ middle vertex on branch for v
19:         else
20:             decide (⊥, 0)                                   ▷ center vertex
21:     else                                                    ▷ branch ≠ ⊥
22:         if ∃v ∈ V s.t. all BRANCH messages have value v then
23:             decide (v, 2)                                   ▷ leaf vertex for v
24:         else
25:             decide (branch,1)                               ▷ middle vertex on branch chosen in round 1
```

or $\perp$. For $R = 2$, processes exchange their branch variables in the second round. The only possible non-$\perp$ value that can be exchanged is v, so the only possible decisions are $(\perp, 0)$, $(v, 1)$ and $(v, 2)$, which proves binding for $R = 2$.

5 Tolerating Malicious Failures

5.1 Lower Bounds

Resiliency. We first note that $n > 3f$ is required to solve connected consensus for any $R \geq 1$ with malicious failures, even without binding. This simple lower bound can be derived from [35]; we provide a complete proof in the full version.

Round Complexity. To show a lower bound on the round complexity, we use the same reduction from approximate agreement as for crash failures in Section 4.1, except that A_{CC} tolerates malicious failures. We then appeal to a result in [20] to show that, as long as $n \leq 9f$, at least two rounds of message exchange are necessary to solve connected consensus when $R = 2$, in the presence of malicious failures.

The relevant result in [20] is that the best the convergence ratio r can be in the presence of malicious failures is $\left\lceil \frac{n-3f}{2f} \right\rceil^{-1}$. Plugging this value of r into the formula $\left\lceil \log_{\frac{1}{r}} \frac{U}{\epsilon} \right\rceil$, with $U = 1$ and $\epsilon = 1/4$, results in a number of rounds that is at least 2 as long as $n \leq 9f$.

If the resilience is sufficiently large, $n > 12f$, a simple one-round connected consensus algorithm is possible for $R = 2$. Furthermore, if $n > 13f$, then the algorithm also satisfies Binding. These algorithms are presented in the full version.

5.2 Algorithms

We next present two algorithms for connected consensus, tolerating malicious failures. We start with a simple algorithm for $n > 5f$ using ideas from [20]. Then we present a more complex algorithm for $n > 3f$, which is a modular extension of algorithms in [3] incorporating new ideas inspired by [34] to deal with multi-valued inputs. Although the $n > 3f$ algorithm has better resilience than the $n > 5f$ algorithm, it has worse time complexity. The round lower bounds discussed above show that it is optimal for $R = 1$ and for $R = 2$, it is optimal as long as $n \leq 9f$.

5.3 Algorithm for $n > 5f$

We now present an algorithm for connected consensus with binding for n processes that tolerates $f < n/5$ malicious failures. The algorithm is extremely simple and efficient, using one round of message exchanges for $R = 1$ and two rounds of message exchanges for $R = 2$, and thus using only $O(n^2)$ messages.

The pseudocode, which is similar to Algorithm 1, appears in Algorithm 2. In the first round of the algorithm, processes exchange their inputs and, after hearing from $n - f$ processes, each process drops the f smallest and f largest values received, an idea inspired by approximate agreement algorithms (e.g., [20]). Then each process chooses branch v of the spider graph, if all the remaining values equal v, or the center vertex otherwise. If $R = 1$, then the process decides on the branch. Otherwise, processes exchange branch values (v or $\perp$) in a second round in order to decide on a vertex on the v branch. This is done in a manner that is similar to the second round in our crash-resilient algorithm, Algorithm 1. After waiting for $n - f$ messages in the second round, if a process' branch is $\perp$, then it decides $(v, 1)$ if at least $f + 1$ second-round messages are for v and $(\perp, 0)$ otherwise. If the process' branch is v, then it decides $(v, 2)$ if at least $n - 2f$ second-round messages are for v and $(v, 1)$ otherwise.

▶ **Theorem 4.** *If $n > 5f$, then Algorithm 2 solves binding connected consensus for $R = 1$ and $R = 2$ with n processes, up to f of which can be malicious. It takes 1 time unit and sends $O(n^2)$ messages for $R = 1$ and takes 2 time units and sends $O(n^2)$ messages for $R = 2$.*

The proof of this theorem appears in the full version. Here, we only outline why the algorithm is binding, which similarly to Algorithm 1 follows from a stronger property, that the branch along which decisions are made is determined solely by the inputs.

Given an assignment of inputs to the processes, suppose there is one execution in which a correct process decides $u \in V$ and another execution in which a correct process decides $v \in V$ with $u < v$. It can be shown that at least $n - 3f$ correct processes have input at most u, and at least $n - 3f$ correct processes have inputs at least v. Thus the total number of processes n is at least $2(n - 3f)$ plus the f faulty processes. That is, $n \geq 2(n - 3f) + f$, which implies $n \leq 5f$, a contradiction. This implies binding for $R = 1$. For $R = 2$, this argument shows that there is only one possible $v \in V$ that can appear in correct processes' branch variables at the end of round 1 in any execution. Thus no correct process can get more than f BRANCH messages for any $u \in V$ other than v and thus it cannot decide $(u, 1)$ or $(u, 2)$ in any execution.

Algorithm 2 Connected Consensus algorithm with Binding for $R = 1, 2$ with n processes, $f < n/5$ of which may be malicious; code for process p.

```
1:  send ⟨INPUT,input⟩ to all                                   ▷ round 1
2:  wait for n − f INPUT messages
3:  let W be multiset of values received in INPUT messages, dropping f smallest and f largest
4:  if ∃v ∈ V s.t. every element in W is v then
5:      branch := v
6:  else
7:      branch := ⊥
8:  if R = 1 then
9:      if branch = ⊥ then
10:         decide (⊥, 0)                                       ▷ center vertex
11:     else
12:         decide (branch,1)                                   ▷ leaf vertex for chosen branch
13: else                                                        ▷ R = 2; round 2
14:     send ⟨BRANCH,branch⟩ to all
15:     wait for n − f BRANCH messages
16:     if branch = ⊥ then
17:         if ∃v ∈ V s.t. at least f + 1 BRANCH messages have value v then
18:             decide (v, 1)                                   ▷ middle vertex on branch for v
19:         else
20:             decide (⊥, 0)                                   ▷ center vertex
21:     else                                                    ▷ branch ≠ ⊥
22:         if ∃v ∈ V s.t. at least n − 2f BRANCH messages have value v then
23:             decide (v, 2)                                   ▷ leaf vertex for v
24:         else
25:             decide (branch,1)                               ▷ middle vertex on branch chosen in round 1
```

5.4 Algorithm for $n > 3f$

We now present an algorithm for connected consensus with binding for n processes that tolerates $f < n/3$ malicious failures. The time complexity for $R = 1$ is 5 and for $R = 2$ is 7, while the message complexity is $O(|V| \cdot n^2)$ in both cases. The failure-resiliency is optimal, per the discussion at the beginning of this section. The $|V|$ factor in the message complexity can be reduced to a constant, resulting in $O(n^2)$ message complexity, by first employing the RD-broadcast primitive in [34], which reduces the number of values under consideration to 6, at the cost of $O(n^2)$ additional messages and two additional time units.

The algorithm is a modular combination of Algorithms 4 (for $R = 1$) and 6 (for $R = 2$) in [3], which work when $|V| = 2$, with the addition of a mechanism from the MV-broadcast in [34] to handle $|V| > 2$. MV-broadcast is a primitive to reduce the number of input values under consideration to two in the context of solving consensus.

Processes exchange input values in increasing "levels" of ECHO messages. Initially, processes exchange their inputs via ECHO messages and also use ECHO messages to amplify values that have been received at least $f + 1$ times; this threshold ensures that at least one correct process has that value as its input. To handle the situation when $|V| > 2$ and there may not be any message that is sent in at least $f + 1$ ECHO messages, an ECHO message for $\perp$ is initiated if a process receives at least $f + 1$ ECHO messages in addition to those for the most commonly received value; this condition only holds if there are at least two different input values at the correct processes. This early appearance of $\perp$ requires some later modifications, mentioned below, to the original algorithm.

Whenever a process receives $n - f$ ECHO messages for some value v, it stores v in its local set variable "approved"; the first time this happens, it sends the value in an ECHO2 message. The $n - f$ threshold ensures some level of uniformity in processes' "approved" sets. Each process sends one ECHO3 message, either for $\perp$ if it collects more than one approved value, or for value v if it receives $n - f$ ECHO2 messages for v. The ECHO3 messages have the desirable property that only one non-$\perp$ value is sent in them by the correct processes. When $R = 1$, processes decide once at least $n - f$ ECHO3 messages have been received: if there are at least $n - f$ for the same value v, then $(v, 1)$ is decided, otherwise if there are at least two approved values or if $\perp$ is approved, then $(\perp, 0)$ is decided. (Checking if $\perp$ has been approved here and later are modifications needed because of the possibility that $\perp$ is sent in ECHO messages.)

When $R = 2$, processes continue for two more levels in order to refine the values obtained so far on the chosen branch. The value chosen as the decision in the $R = 1$ case is sent in an ECHO4 message; these message inherit the property that at most one non-$\perp$ appears in those sent by correct processes. Each process waits for ECHO4 messages. If eventually it collects $n - f$ for a common value, then it sends an ECHO5 message for that value; if eventually it has more than one approved value or if $\perp$ is approved, it sends an ECHO5 message for $\perp$. ECHO5 messages also inherit the property that at most one non-$\perp$ appears in those sent by correct processes.

The decision is based on ECHO4 and ECHO5 messages received. If a process receives at least $n - f$ ECHO5 messages for the same non-$\perp$ value, then it decides $(v, 2)$. If it receives at least $n - f$ ECHO5 messages for $\perp$, then it decides $(\perp, 0)$. Otherwise, if it has approved either $\perp$ or at least two values, receives at least one ECHO5 message for some non-$\perp$ value w, and has at least $f + 1$ ECHO4 messages for w, it decides $(w, 1)$.

The pseudocode is in Algorithm 3. The presentation differs from that of our Algorithms 1 and 2 and of the algorithms in [3]. Instead of using syntactic constructs such as "wait until" and "upon" receiving certain messages, our code is purely interrupt-driven in order to clarify the interactions between the receipts of different messages and the conditions triggering various actions. The technique inspired by [34] appears in Lines 14 through 15. We denote by $\text{sum}(A)$, where A is an array of integers, the sum of all the entries in A. A correct process sends at most one ECHO message for any $v \in V \cup \{\perp\}$, and at most one ECHO2, ECHO3, ECHO4, and ECHO5 message. This allows us to assume there is some mechanism for eliminating duplicate messages that arrive from the same (faulty) sender.

▶ **Theorem 5.** *If $n > 3f$, then Algorithm 3 solves binding connected consensus for $R = 1$ and $R = 2$ with n processes, up to f of which can be malicious. It takes 5 time units and sends $O(|V| \cdot n^2)$ messages for $R = 1$ and takes 7 time units and sends $O(|V| \cdot n^2)$ messages for $R = 2$.*

The complete proof of Theorem 5 appears in the full version and is sketched below.

Proof (Sketch). First, we argue that the values sent in ECHO messages by correct processes are "valid": if the value is in V, then some correct process has input v, whereas if the value is $\perp$, then not all the correct processes have the same input. The latter property is ensured by lines 14 through 15 for the following reason. The first correct process to send ECHO for $\perp$ does so because it receives at least $f + 1$ ECHO messages for values other than the most frequently occurring value m of the ECHO messages it has received so far. Letting x be the number of ECHO messages received for m, it follows that at least $x + 1$ of the ECHO messages for values other than m are from correct processes. But they cannot all be for the same value since no value occurs more frequently than m.

Algorithm 3 Connected Consensus algorithm with Binding for $R = 1, 2$ with n processes, $f < n/3$ of which may be malicious; code for process p.

initially:
approved := $\emptyset$ (subset of $V \cup \{\perp\}$) ▷ *set of approved values*
num_echo[v] := 0 for $v \in V \cup \{\perp\}$ ▷ *# received* ECHO *messages for v*
num_echo$i$[v] := 0 for $2 \le i \le 5$, $v \in V \cup \{\perp\}$ ▷ *# received* ECHO-i *messages for v*
sent_echo[v] := false for $v \in V \cup \{\perp\}$ ▷ *has p sent an* ECHO *message for v yet?*
sent_echo$i$:= false for $2 \le i \le 5$ ▷ *has p sent an* ECHO-i *message yet?*
decided := false ▷ *has p decided yet?*

wakeup:
send ⟨ECHO,input⟩ to all; sent_echo[input] := true ▷ *initiate* ECHO *for p's input*

receive ⟨ECHO, v⟩:
num_echo[v] + +
if (num_echo[v] = $f + 1$) and (!sent_echo[v]) **then**
  send ⟨ECHO, v⟩ to all; sent_echo[v] := true ▷ *echo v if enough support but only once*
else if (sum(num_echo) − num_echo[m] ≥ $f + 1$) and (!sent_echo[$\perp$]),
where m is s.t. num_echo[m] ≥ num_echo[u] for all $u \in V \cup \{\perp\}$) **then**
  ▷ *if evidence for multiple correct inputs then initiate* ECHO *for* $\perp$
  send ⟨ECHO, $\perp$⟩ to all; sent_echo[$\perp$] := true
else if num_echo[v] = $n - f$ **then**
  if !sent_echo2 **then** ▷ *send only one* ECHO2
    send ⟨ECHO2, v⟩ to all; sent_echo2 := true
  add v to approved
  if (|approved| > 1) and (!sent_echo3) **then** ▷ *send only one* ECHO3
    send ⟨ECHO3, $\perp$⟩ to all; sent_echo3 := true

receive ⟨ECHO2, v⟩:
num_echo2[v] + +
if (num_echo2[v] = $n - f$) and (!sent_echo3) **then** ▷ *send only one* ECHO3
  send ⟨ECHO3, v⟩ to all; sent_echo3 := true

receive ⟨ECHO3, v⟩:
num_echo3[v] + +
if (sum(num_echo3) ≥ $n - f$) and ((|approved| > 1) or ($\perp \in$ approved)) **then**
  if ($R = 1$) and (!decided) **then** ▷ *decide only once*
    decide ($\perp$, 0); decided := true ▷ *center vertex*
  else if ($R = 2$) and (!sent_echo4) **then** ▷ *send only one* ECHO4
    send ⟨ECHO4, $\perp$⟩ to all; sent_echo4 := true
else if num_echo3[v] ≥ $n - f$ **then**
  if ($R = 1$) and (!decided) **then** ▷ *decide only once*
    decide (v, 1); decided := true ▷ *leaf vertex for v*
  else if ($R = 2$) and (!sent_echo4) **then** ▷ *send only one* ECHO4
    send ⟨ECHO4,v⟩ to all; sent_echo4 := true

▷ *continued.....*

▷ .....*Continuation of Algorithm 3*

38: receive $\langle$ECHO4$, v\rangle$:
39: num_echo4$[v]++$
40: **if** (num_echo4$[v] = n - f$) and (!sent_echo5) **then** ▷ *send only one* ECHO5
41: send $\langle$ECHO5$, v\rangle$ to all; sent_echo5 := true
42: **else if** (sum(num_echo4) $\geq n - f$) and ((|approved| > 1) or ($\perp \in$ approved)) and (!sent_echo5) **then**
43: send $\langle$ECHO5$, \perp\rangle$ to all; sent_echo5 := true

44: receive $\langle$ECHO5$, v\rangle$:
45: num_echo5$[v]++$
46: **if** !decided **then** ▷ *decide only once*
47: **if** ($v \in V$) and (num_echo5$[v] \geq n - f$) **then**
48: decide $(v, 2)$; decided := true ▷ *leaf vertex for* v
49: **else if** (sum(num_echo5) $\geq n - f$) and ((|approved| > 1) or ($\perp \in$ approved)) and there exists $w \in V$ s.t. (num_echo5$[w] \geq 1$) and (num_echo4$[w] \geq f + 1$) **then**
50: decide $(w, 1)$; decided := true ▷ *middle vertex on branch for* w
51: **else if** num_echo5$[\perp] \geq n - f$ **then**
52: decide $(\perp, 0)$ ▷ *center vertex*

Since a correct process approves a value when it gets $n - f$ ECHO messages for it, the validity of the ECHO messages implies validity of the approved values. In addition, the approved sets of all the correct processes are eventually the same. The ECHO2 messages preserve the validity properties of the ECHO messages and also ensure that the values sent in them end up being approved. The ECHO3 messages add a "uniqueness" property, ensuring at most one non-$\perp$ value is sent by correct processes. They also satisfy a modified approval property: if v is sent by a correct process and v is not $\perp$, then eventually every correct process approves v, otherwise (if $v = \perp$) eventually every correct process either approves $\perp$ or approves at least two values.

We can now prove correctness and complexity when $R = 1$. *Validity* is immediate from the validity properties ensured by the ECHO* messages. *Agreement* follows from *Binding* (cf. Proposition 1), which we discuss next. The first correct process to decide receives $n - f$ ECHO3 messages, at least $n - 2f$ of which are from correct processes. If any of these messages are for a value in V, then by uniqueness of ECHO3 messages, no correct process can send an ECHO3 message for a different non-$\perp$ value, and thus no such value can be decided subsequently. If all of these messages are for $\perp$, then other processes can receive at most $2f$ ECHO3 messages for any $v \in V$ (f from the correct processes that did not send ECHO3 for $\perp$ and f from the faulty processes), which is not enough support for deciding v subsequently.

We next address *Termination and time complexity.* In the full correctness proof, we first show that the algorithm terminates, before we scale all the message delays by the duration of the longest one. We first show that every correct process sends an ECHO2 message by time 2. If at least $f + 1$ correct inputs are the same, say v, then by time 1, every correct process receives the initial ECHO messages for v, and sends an ECHO message for v if it has not already done so. Thus every correct process receives at least $n - f$ ECHO messages by time 2, and sends ECHO2.

The more interesting case, which only arises when $|V| > 2$, is when no value occurs at least $f + 1$ times among the inputs of the correct processes. Let x_i (resp., y_i) be the number of ECHO messages for v_i received by a correct process p from correct (resp., faulty) processes

by time 1, $1 \leq i \leq |V|$. Note that $x_i \leq f$ and that $x_1 + \ldots + x_{|V|} \geq n - f$ since p has received all the ECHO messages sent by the correct processes initially. Let v_m be the value that occurs most frequently among all the ECHO messages received by time 1 (breaking ties arbitrarily). Then the total number of ECHO messages minus the number of those for v_m is

$$\left(\sum_{i=1}^{|V|}(x_i + y_i)\right) - (x_m + y_m) = \sum_{i=1}^{|V|} x_i + \sum_{\substack{i=1 \\ i \neq m}}^{|V|} y_i - x_m \geq (n - f) - f \geq f + 1,$$

since $n > 3f$. Thus p sends an ECHO message for $\perp$ by time 1 if it hasn't already done so, and so by time 2, every correct process receives $n - f$ ECHO messages for $\perp$ and sends ECHO2 for $\perp$ if it has not already sent an ECHO2 message.

We next show that every correct process p sends an ECHO3 message by time 4. We rely on the fact that if v is in a correct process' approved set at time t, then every correct process approved set by time $t + 2$, which implies that if v is sent by a correct process in an ECHO2 message, then every correct process approves v by time 4. Since ECHO2 messages are sent by correct processes by time 2, at least $n - f$ arrive at p by time 3. If they are all for a common value v, then p sends ECHO3 for v by time 3. Otherwise, p waits until it has at least two approved values. Process p must have received an ECHO2 for value v_1 from a correct process and an ECHO2 for a different value v_2 from a different correct process. Thus p approves v_1 and v_2 by time 4 and sends ECHO3 by time 4.

We use a similar argument to the previous paragraph to show that every correct process p decides by time 5. It relies on the key fact that the approval of values sent in ECHO3 messages by the correct processes takes place by time 5. Thus p either receives at least $n - f$ ECHO3 messages for a common value by time 5 or has approved either $\perp$ or at least two values by time 5, and thus decides by time 5.

The *message complexity* is $O(|V| \cdot n^2)$ since each correct process sends to all the processes at most one ECHO message for each $v \in V$, one ECHO2 message, and one ECHO3 message.

We continue to prove correctness and complexity when $R = 2$. First note that when $R = 2$, a process sends an ECHO4 message for v under exactly the same circumstances that it decides $(v, 1)$ (if $v \in V$) or $(v, 0)$ (if $v = \perp$) when $R = 1$. Thus we have the analogous properties for ECHO4 messages that we had for ECHO3 messages (validity, uniqueness, and approval). These properties also carry over to ECHO5 messages.

For *Agreement*, we first recall that all the non-$\perp$ values sent in ECHO4 and ECHO5 messages are the same, call it v. It remains to show that if a correct process p decides $(v, 2)$ then no correct process can decide $(\perp, 0)$. Since p decides $(v, 2)$, it receives $n - f$ ECHO5 message for v. But since $n > 3f$, it's not possible for another process to receive $n - f$ ECHO5 messages for $\perp$, which is required for a decision of $(\perp, 0)$.

Validity follows from the validity properties of ECHO4 and ECHO5 messages.

Binding holds since the binding property for $R = 1$ implies that there is only one possible non-$\perp$ value that can be decided in any extension after the first correct process sends its ECHO4 message.

For *Termination and time complexity*, we prove every correct process decides by time 7.

Since ECHO4 messages are sent for $R = 2$ exactly when decisions are made for $R = 1$, we know that correct processes send ECHO4 by time 5. Fortunately, the approval time of 5 for values in ECHO3 messages carries over to ECHO4 messages. Thus p either receives at least $n - f$ ECHO4 messages for a common value by time 6 or has approved either $\perp$ or at least two values by time 5, and thus sends ECHO5 by time 6.

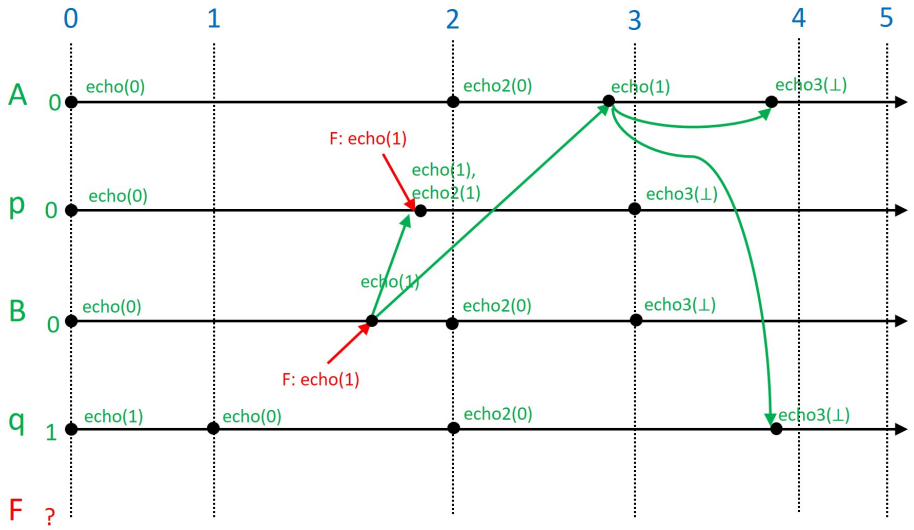


Figure 4 Diagram illustrating scenario in which decision is delayed until almost time 5.

Similarly, the approval time of 5 also holds for values in ECHO5 messages. Thus p either receives at least $n - f$ ECHO5 messages for a common value by time 7 and decides, or has approved either $\perp$ or at least two values by time 5. In the latter case, since p receives less than $n - f$ ECHO5 messages for $\perp$, it receives an ECHO5 message for some $w \in V$ from a correct process q. In turn, q received at least $n - f$ ECHO4 messages for w, at least $n - 2f \geq f + 1$ of which are from correct processes. Since these correct processes send their ECHO4 messages for w by time 5, p receives them by time 6. Thus p decides by time 7.

The *message complexity* is still $O(|V| \cdot n^2)$ since in addition to the messages sent when $R = 1$, each process sends to all processes one ECHO4 message and one ECHO5 message. ◀

Time complexity versus round complexity. The upper bounds of 5 and 7 on the *time complexities* for Algorithm 3 are tight, as shown by an execution described below; see more details in the full version. The execution uses $V = \{0, 1\}$ and thus it is also an execution of Algorithm 4 in [3], implying that the tight time complexity of the latter algorithm is also 5, and that of Algorithm 6 in [3] is 7. This is in contrast to the *round complexities* of 4 and 6 calculated in [3] for their Algorithms 4 and 6.

The discrepancy between round complexity and time complexity is caused by the waiting conditions imposed before performing the next broadcast. If the condition is simply to receive enough messages from the previous broadcast, then at most one time unit elapses per broadcast. But if there is an additional condition, for instance, waiting to approve at least two values, then the condition may take more than one time unit to become true. We next sketch the execution to illustrate this point; see Figure 4.

Assume $n = 3f + 1$, where $f \geq 2$, and partition the correct processes into sets A, B, $\{p\}$, and $\{q\}$, where $|A| = f - 1$, $|B| = f$, and let F be the set of f faulty processes. Initially all correct processes have input 0 except q has input 1. At time 0, processes send ECHO messages with their inputs which arrive at time 1. At time 1, q sends an ECHO message for 0, which arrives at time 2. Just before time 2, every process in B receives f ECHO messages for 1 from the processes in F, which causes it to send ECHO for 1. Those messages take 1 time unit to arrive at all processes except p, which receives them immediately. This then causes p to send an ECHO message for 1. Immediately thereafter, p receives f ECHO messages for 1

from the processes in F, which causes it to send its ECHO2 message, for 1. These messages from p take 1 time unit to arrive. Then at time 2, all the correct processes send ECHO2 message for 0, except for p, which has already sent its ECHO2 message (for 1). The difficulty is that by time 3, no process has $n - f$ ECHO2 messages for a common value, due to the ECHO2 message for 1 from p. The processes in $B \cup \{p\}$ are not blocked from sending ECHO3 because they have approved both 0 and 1, but the processes in $A \cup \{q\}$ have only approved 0; they are unable to approve 1 until they get the ECHO messages for 1 sent by the processes in $A \cup \{q\}$, which does not happen until just before time 4. Letting all the ECHO3 messages have delay 1 means that processes cannot decide until shortly before time 5.

6 Discussion

We have proposed a new problem called connected consensus which generalizes a number of primitives used to solve consensus, including crusader agreement, graded broadcast, and adopt-commit, using a numeric parameter R. The problem can be reduced to real-valued approximate agreement when the input set is binary and and to approximate agreement on graphs for multi-valued input sets (two or more inputs). We extended the definition of the binding property for such primitives to the multi-valued case.

We presented efficient message-passing algorithms for connected consensus when R is 1 (corresponding to crusader agreement) or 2 (corresponding to graded broadcast), in the presence of crash and malicious failures, for multi-valued input sets.

Our algorithm for crash failures has optimal resilience and message complexity; its time complexity is optimal for reasonable resiliencies and improves on the best previously known algorithms, which only handled binary inputs.

For malicious failures, we provide two algorithms that trade off resilience against time and message complexity. One algorithm has time complexity 1 or 2 (for $R = 1$ or $R = 2$) and sends $O(n^2)$ messages, but requires $n > 5f$. The other algorithm only requires $n > 3f$, but has time complexity 5 or 7 (for $R = 1$ or $R = 2$) and sends $O(|V| \cdot n^2)$ messages. This is the same performance as the algorithms in [3] which are only for the case when $|V| = 2$.

The techniques used in our (simple) algorithms for crash failures and for malicious failures with $n > 5f$ are familiar from prior work. For example, algorithms in [12, 26, 33] rely on similar mechanisms to solve (standard) consensus using various kinds of oracles. The novelty in our work is the focus on the binding property for a key subproblem of consensus, extracted as connected consensus.

There is a message-passing algorithm for adopt-commit with multi-valued inputs that works in the presence of malicious failures as long as $n > 3f$ [11]. However, the number of possible inputs must be smaller than $\lfloor \frac{n-(f+1)}{f} \rfloor$, while our algorithm works for any size input set. The message complexity is $O(n^3)$ as compared to our $O(|V| \cdot n^2)$. Furthermore, this algorithm avoids the challenge of ensuring the binding property (which anyway is not well-defined for adopt-commit) as it is combined with an oracle in order to solve consensus.

Concurrent work by Abraham, Ben-David, Stern and Yandamuri appearing in these proceedings [1] studies the round complexity of binary (graded) crusader agreement both with and without binding. A key difference is in the definition of the adversary: In [1] it is assumed that the adversary can adaptively choose the inputs of the processes after the start of the execution when the processes take their first steps. In contrast, in our model, the adversary does not have that power, and the inputs are fixed at the beginning of each execution. As a result, some of the lower bounds in [1] are larger than some of our upper bounds. It is argued in [1] that tolerating such a strong adversary can be advantageous for developing simple and efficient randomized consensus algorithms.

An intriguing open question is whether there is an inherent cost for satisfying the binding property: is there some measure, perhaps time, in which solving connected consensus without binding is more efficient than solving it with binding?

Adopt-commit and related primitives have been implemented also in shared-memory systems, e.g., [6, 18, 27, 32]. The connection we have made between connected consensus and approximate agreement (on graphs) may contribute to finding improved algorithms for these primitives in shared memory.

This connection might also be a fruitful direction for future work on connected consensus in other timing models. For instance, [5] uses adopt-commit (and variants) to solve consensus in eventually synchronous systems. Another interesting direction is to study connected consensus in other fault models, such as the authenticated setting; authenticated algorithms appear in [3] but they are for binary inputs.

References

1 I. Abraham, N. Ben-David, G. Stern, and S. Yandamuri. On the round complexity of asynchronous crusader agreement. In *OPODIS*, 2023.

2 Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In *OPODIS*, pages 229–239. Springer, 2004. doi:10.1007/11516798_17.

3 Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In *41st ACM Symposium on Principles of Distributed Computing*, pages 381–391, 2022. doi:10.1145/3519270.3538426.

4 Yehuda Afek, James Aspnes, Edo Cohen, and Danny Vainstein. Brief announcement: Object oriented consensus. In *36th ACM Symposium on Principles of Distributed Computing*, pages 367–369, 2017. Full version in https://www.cs.yale.edu/homes/aspnes/papers/vac-abstract.html. doi:10.1145/3087801.3087867.

5 Karolos Antoniadis, Julien Benhaim, Antoine Desjardins, Elias Poroma, Vincent Gramoli, Rachid Guerraoui, Gauthier Voron, and Igor Zablotchi. Leaderless consensus. *Journal of Parallel and Distributed Computing*, 176:95–113, 2023. doi:10.1016/J.JPDC.2023.01.009.

6 James Aspnes. Faster randomized consensus with an oblivious adversary. In *31st ACM Symposium on Principles of Distributed Computing*, pages 1–8, 2012. doi:10.1145/2332432.2332434.

7 Hagit Attiya, Constantin Enea, and Shafik Nassar. Faithful simulation of randomized BFT protocols on block DAGs. In *Concur*, 2023. URL: https://eprint.iacr.org/2023/192.

8 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill Publishing Company, 1st edition, 1998.

9 Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16:249–262, 2003. doi:10.1007/S00446-002-0083-3.

10 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous Byzantine agreement with subquadratic communication. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18*, pages 353–380. Springer, 2020. doi:10.1007/978-3-030-64375-1_13.

11 Zohir Bouzid, Achour Mostefaoui, and Michel Raynal. Minimal synchrony for Byzantine consensus. In *34th ACM Symposium on Principles of Distributed Computing*, pages 461–470, 2015. doi:10.1145/2767386.2767418.

12 Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *6th International Conference on Parallel Computing Technologies*, volume 2127, pages 42–50, 2001. doi:10.1007/3-540-44743-1_4.

13 Armando Castañeda, Sergio Rajsbaum, and Matthieu Roy. Convergence and covering on graphs for wait-free robots. *Journal of the Brazilian Computer Society*, 24:1–15, 2018. doi:10.1186/S13173-017-0065-8.

14 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

15 Ran Cohen, Pouyan Forghani, Juan Garay, Rutvik Patel, and Vassilis Zikas. Concurrent asynchronous Byzantine agreement in expected-constant rounds, revisited. Cryptology ePrint Archive, Paper 2023/1003, 2023. URL: https://eprint.iacr.org/2023/1003.

16 Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006. doi:10.1093/COMJNL/BXH145.

17 Giovanni Deligios, Martin Hirt, and Chen-Da Liu-Zhang. Round-efficient Byzantine agreement and multi-party computation with asynchronous fallback. In *19th International Conference on Theory of Cryptography, TCC*, pages 623–653, 2021. doi:10.1007/978-3-030-90459-3_21.

18 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. *Journal of Parallel and Distributed Computing*, 153:110–118, 2021. doi:10.1016/J.JPDC.2021.03.015.

19 Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

20 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, 1986. doi:10.1145/5925.5931.

21 Alan David Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4:9–29, 1990. doi:10.1007/BF01783662.

22 Alan David Fekete. Asynchronous approximate agreement. *Information and Computation*, 115(1):95–124, 1994. doi:10.1006/INCO.1994.1094.

23 Paul Feldman and Silvio Micali. Optimal algorithms for Byzantine agreement. In *12th Annual ACM Symposium on Theory of Computing*, pages 148–161, 1988. doi:10.1145/62212.62225.

24 Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous Byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997. doi:10.1137/S0097539790187084.

25 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.

26 Roy Friedman, Achour Mostéfaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Trans. Dependable Secur. Comput.*, 2(1):46–56, 2005. doi:10.1109/TDSC.2005.13.

27 Eli Gafni. Round-by-round fault detectors: unifying synchrony and asynchrony. In *17th ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998. doi:10.1145/277697.277724.

28 Manfred Koebe. On a new class of intersection graphs. In *Annals of Discrete Mathematics*, volume 51, pages 141–143. Elsevier, 1992. doi:10.1016/S0167-5060(08)70618-6.

29 Stephen R Mahaney and Fred B Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *4th ACM Symposium on Principles of Distributed Computing*, pages 237–249, 1985. doi:10.1145/323596.323618.

30 Atsuki Momose and Ling Ren. Constant latency in sleepy consensus. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2295–2308, 2022. doi:10.1145/3548606.3559347.

31 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *J. ACM*, 62(4):31:1–31:21, 2015. doi:10.1145/2785953.

32 Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The combined power of conditions and information on failures to solve asynchronous set agreement. *SIAM J. on Computing*, 38(4):1574–1601, 2008. doi:10.1137/050645580.

33 Achour Mostéfaoui and Michel Raynal. Leader-based consensus. *Parallel Process. Lett.*, 11(1):95–107, 2001. doi:10.1142/S0129626401000452.

34 Achour Mostéfaoui and Michel Raynal. Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Acta Informatica*, 54(5):501–520, 2017. doi:10.1007/s00236-016-0269-y.

35 Thomas Nowak and Joel Rybicki. Byzantine approximate agreement on graphs. In *33rd International Symposium on Distributed Computing*, pages 29:1–29:17, 2019. doi:10.4230/LIPICS.DISC.2019.29.

36 Sam Toueg. Randomized Byzantine agreements. In *3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, 1984. doi:10.1145/800222.806744.

37 Jiong Yang, Gil Neiger, and Eli Gafni. Structured derivations of consensus algorithms for failure detectors. In *17th ACM Symposium on Principles of Distributed Computing*, pages 297–306, 1998. doi:10.1145/277697.277755.

A No Built-in Binding, for $n < 5f$

In this section we show that the locked value for the Binding property cannot be predetermined from the correct processes' inputs if $n \leq 5f$, even if faulty processes cannot equivocate and the input set is binary. Our approach is to consider any algorithm that works by having processes obtain $n - f$ values, at most one from each process, such that if two processes p and q both get values corresponding to process r, then the values are the same and if r is correct then that value is r's input. Then the algorithm must decide based only on the multiset of values it has received. We call such an algorithm *uniform*.

Consider a uniform algorithm with $n \leq 5f$. We show that if a process receives all but f 0's, then it must decide 0 and if it receives all but f 1's, then it must decide 1. Then we show that if a process receives about half 0's and half 1's, then it must decide $(\perp, 0)$ (this relies on the fact that two configurations with different non-$\perp$ values must have more than f values that are different). Finally we rely on the fact that the number of values a process receives is at most $4f$ to show that if the first process to decide gets about half 0's and half 1's and decides $(\perp, 0)$, then in one extension we can replace f of the 0's with 1's to get a configuration that requires a decision of $(1, 1)$, and in another extension we can replace f of the 1's with 0's to get a configuration that requires a decision of $(0, 1)$, which violates Binding.

▶ **Theorem 6.** *If a uniform connected consensus algorithm for $R = 1$ with n processes, up to f of which can be malicious, satifies the Binding property, then $n > 5f$.*

Proof. Consider for contradiction such an algorithm with $n \leq 5f$. Without loss of generality, assume $V = \{0, 1\}$. For simplicity, we refer to the possible decisions as 0, 1, and $\perp$, instead of $(0, 1)$, $(1, 1)$, and $(\perp, 0)$. Let $D(z) \in \{0, 1, \perp\}$ denote the decision made when the multiset of $n - f$ values received has z 0's.

We first show that if there is an "overwhelming" number of 0's received, then the decision must be 0, and similarly for 1. The threshold is $n - 2f$, which is at least $f + 1$ since the resilience lower bound discussed in Section 5.1 shows that n must be at least $3f + 1$.

▶ **Lemma 7.** *Let z be any integer in $\{0, \ldots, n - 2f\}$.*
(a) *If $z \geq n - 2f$, then $D(z) = 0$.*
(b) *If $z \leq f$, then $D(z) = 1$.*

Proof.
(a) Consider any execution in which correct process p receives $z \geq n-2f$ 0's and $n-f-z$ 1's. This execution is indistinguishable from one in which all $n - f$ of the correct processes have input 0, and p receives $z \geq n - 2f$ messages for 0 from the correct processes and $n - f - z \leq f$ messages for 1 from faulty processes. By the Validity condition, p must decide 0 in the second execution. Thus $D(z) = 0$.

(b) Consider any execution in which correct process p receives $z \leq f$ 0's and $n - f - z$ 1's. This execution is indistinguishable from one in which all $n - f$ of the correct processes have input 1, and p receives $n - f - z$ messages for 1 from the correct processes and $z \leq f$ messages for 0 from faulty processes. By the Validity condition, p must decide 1 in the second execution. Thus $D(z) = 1$. ◀

▶ **Lemma 8.** *The number of processes n must be at least $3f + 2$.*

Proof. Suppose in contradiction that $n = 3f + 1$ and consider any correct process p. If the majority of the $n - f = 2f + 1$ values received by p is 0, then $z \geq f + 1$. Since $n = 3f + 1$, $f + 1 = n - 2f$, and thus Lemma 7(a) implies that p must decide 0. If the majority value is not 0, then the majority value must be 1, and Lemma 7(b) implies that p must decide 1. Thus there is no possibility of a process deciding $\perp$, and hence the algorithm actually solves consensus, which is impossible [25]. ◀

The next lemma shows that the range of input values requiring a decision of 0 and the range of input values requiring a decision of 1 must be sufficiently separated from each other.

▶ **Lemma 9.** *Let x and y be integers in $\{0, \ldots, n - 2f\}$. If $D(x) = 0$ and $D(y) = 1$, then $|x - y| > f$.*

Proof. Suppose in contradiction there exist x and y in $\{0, \ldots, n - 2f\}$ such that $D(x) = 0$, $D(y) = 1$, and $|x - y| \leq f$.

Without loss of generality, suppose $x > y$. Consider the execution in which the correct inputs are x 0's and $n - f - x$ 1's. Suppose that correct process p hears from all the correct processes, so it gets x 0's and $n - f - x$ 1's, and decides 0. Suppose that another correct process q hears from only y of the correct processes with input 0, all $n - f - x$ of the correct processes with input 1, and $x - y \leq f$ faulty processes, who pretend to have input 1. Thus q gets y 0's and $(n - f - x) + (x - y) = n - f - y$ 1's, and decides 1. This is possible because of the asynchrony of the message deliveries. But this violates the Agreement property. ◀

The next claim states that when the received values are about half 0's and half 1's, the decision must be $\perp$. It also shows that this situation is not that far from a situation requiring a decision of 0 and also not that far from a situation requiring a decision of 1.

▷ Claim 10. Let $m = \left\lceil \frac{n-f}{2} \right\rceil$.
(a) $m + f \geq n - 2f$,
(b) $m - f \leq f$, and
(c) $D(m) = \perp$.

We show that Binding is not guaranteed.

Consider an execution α in which $m = \left\lceil \frac{n-f}{2} \right\rceil$ of the correct processes have input 0 and the remaining $n - f - m$ correct processes have input 1. Let correct process p be the first to receive $n - f$ messages, which are all from the correct processes. So p gets m 0's and $n - f - m$ 1's. By Claim 10(c), it decides $\perp$.

Also suppose that all messages to another correct process q are delayed until after p decides and that the faulty processes do nothing until after p decides.

Let α_0 be an extension of α in which q receives m 0's from correct processes, $n - 2f - m$ 1's from correct processes, and f 0's from faulty processes. So q receives $m + f$ 0's and $n - 2f - m$ 1's. Since $m + f \geq n - 2f$ by Claim 10(a), Lemma 7(a) implies that q decides 0.

Let α_1 be an extension of α in which q receives $m-f$ 0's from correct processes, $n-f-m$ 1's from correct processes, and f 1's from faulty processes. So q receives $m-f$ 0's and $n-m$ 1's. Since $m-f \leq f$ by Claim 10(b), Lemma 7(b) implies that q decides 1.

The existence of extensions α_1 and α_0 violates the Binding property. ◀

Energy-Constrained Programmable Matter Under Unfair Adversaries

Jamison W. Weber
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Tishya Chhabra
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Andréa W. Richa
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

Individual modules of *programmable matter* participate in their system's collective behavior by expending energy to perform actions. However, not all modules may have access to the external energy source powering the system, necessitating a local and distributed strategy for supplying energy to modules. In this work, we present a general *energy distribution framework* for the *canonical amoebot model* of programmable matter that transforms energy-agnostic algorithms into energy-constrained ones with equivalent behavior and an $\mathcal{O}(n^2)$-round runtime overhead – even under an *unfair adversary* – provided the original algorithms satisfy certain conventions. We then prove that existing amoebot algorithms for *leader election* (ICDCN 2023) and *shape formation* (Distributed Computing, 2023) are compatible with this framework and show simulations of their energy-constrained counterparts, demonstrating how other unfair algorithms can be generalized to the energy-constrained setting with relatively little effort. Finally, we show that our energy distribution framework can be composed with the *concurrency control framework* for amoebot algorithms (Distributed Computing, 2023), allowing algorithm designers to focus on the simpler energy-agnostic, sequential setting but gain the general applicability of energy-constrained, asynchronous correctness.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Self-organization

Keywords and phrases Programmable matter, amoebot model, energy distribution, concurrency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.7

Related Version All proofs omitted from this paper due to space constraints can be found in: *Full Version*: `https://arxiv.org/abs/2309.04898`

Supplementary Material *Software (AmoebotSim)*: `https://github.com/SOPSLab/AmoebotSim` archived at `swh:1:dir:af373f8c717556382caba4a3abccc6ad749f03c3`

Funding This work is supported in part by National Science Foundation award CCF-2312537 and by Army Research Office MURI award #W911NF-19-1-0233.

1 Introduction

Programmable matter [34] is often envisioned as a material composed of simple, homogeneous modules that collectively change the system's physical properties based on environmental stimuli or user input. These modules participate in the system's overall collective behavior by expending energy to perform internal computation, communicate with their neighbors, and move. But as the number of modules per collective increases and individual modules

are miniaturized from the centimeter/millimeter-scale [20, 22, 32] to the micro- and nano-scale [4, 16, 26], traditional methods of robotic power supply such as internal battery storage and tethering become infeasible. Many programmable matter systems instead make use of an external energy source accessible by at least one module and rely on *module-to-module power transfer* to supply the system with energy [6, 20, 23, 32]. This external energy can be supplied directly to modules in the form of electricity [20] or may be ambiently available as light, heat, sound, or chemical energy in the environment [27, 30]. Since energy may not be uniformly accessible to all modules in the system, a strategy for *energy distribution* – sharing energy among modules such that the system can achieve its desired function – is imperative.

Algorithmic theory for programmable matter – including population protocols [1], the nubot model [36], mobile robots [17], hybrid programmable matter [21], and the amoebot model [10, 12] – has largely ignored energy constraints, focusing instead on characterizing individual modules' necessary and sufficient capabilities for goal collective behaviors. Besides a few notable exceptions [16, 32], this literature only references energy to justify assumptions (e.g., why a system should remain connected [28]) and ignores the impact of energy usage and distribution on an algorithm's efficiency. In contrast, both programmable matter practitioners and the modular and swarm robotics literature incorporate energy constraints as influential aspects of algorithm design [2, 24, 29, 31, 35].

This gap motivated the prior Energy-Sharing algorithm for energy distribution [11] under the *amoebot model* of programmable matter [12]. When amoebots do not move and are activated *sequentially and fairly*, Energy-Sharing distributes any necessary energy to all n amoebots within at most $\mathcal{O}(n)$ rounds. Combined with the Forest-Prune-Repair algorithm introduced in the same work to repair energy distribution networks as amoebots move, it was suggested that any amoebot algorithm could be composed with these two to handle energy constraints, though this was only shown for one algorithm in simulation.

In this work, we introduce a general *energy distribution framework* that provably converts any energy-agnostic amoebot algorithm satisfying certain conventions into an *energy-constrained* version that exhibits the same system behavior while also distributing the energy amoebots need to meet the demands of their actions. In particular, we use the message passing-based *canonical amoebot model* [10] to address the challenges of *unfair* adversarial schedulers – the most general of all fairness assumptions – that can activate any amoebot that is able to perform an action regardless of how long others have been waiting to do the same. Under an unfair adversary, the prior Forest-Prune-Repair algorithm may not terminate, rendering it unusable for maintaining energy distribution networks. In contrast, energy-constrained algorithms produced by our framework not only terminate despite unfairness, but do so within an $\mathcal{O}(n^2)$-round overhead, where n is the number of amoebots in the system.

Our Contributions. We summarize our contributions as follows. We introduce the *energy distribution framework* that transforms any energy-agnostic amoebot algorithm $\mathcal{A}$ satisfying some basic conventions and a demand function δ specifying its energy costs into an energy-constrained algorithm $\mathcal{A}^\delta$ that provably exhibits equivalent behavior to $\mathcal{A}$, even under an unfair adversary, while incurring at most an $\mathcal{O}(n^2)$-round runtime overhead (Section 3). We then prove that both the Leader-Election-by-Erosion algorithm from [5] and the Hexagon-Formation algorithm from [10] satisfy the framework's conventions and show simulations of their energy-constrained counterparts produced by the framework (Section 4).

Finally, we prove that a particular class of "expansion-corresponding" algorithms that are compatible with the established *concurrency control framework* for amoebot algorithms [10] – including Leader-Election-by-Erosion and Hexagon-Formation– remain so after transformation

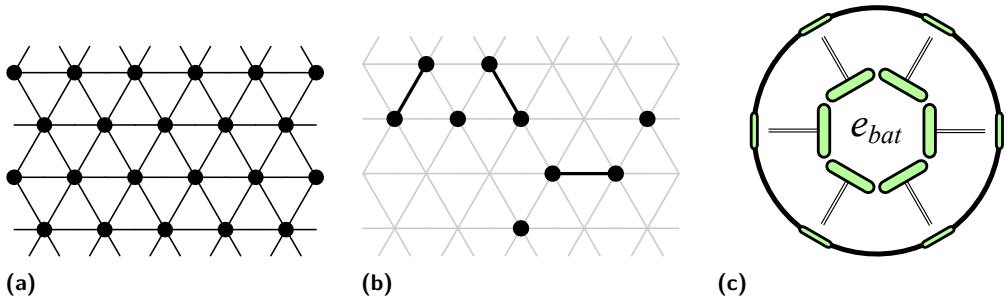


Figure 1 *The Amoebot Model.* (a) A section of the triangular lattice G_Δ used in the geometric space variant; nodes of V are shown as black circles and edges of E are shown as black lines. (b) Expanded and contracted amoebots; G_Δ is shown in gray and amoebots are shown as black circles. Amoebots with a black line between their nodes are expanded. (c) When modeling energy, each amoebot A has a battery $A.e_{bat}$ storing energy for its own use and for sharing with its neighbors.

by our energy distribution framework, establishing a general pipeline for lifting energy-agnostic, non-concurrent amoebot algorithms (which are easier to design and analyze) to the more realistic *energy-constrained, asynchronous setting* (Section 5).

2 Preliminaries

We begin with necessary background on the (canonical) amoebot model in Section 2.1 and our extensions for energy constraints in Section 2.2.

2.1 The Amoebot Model

In the *canonical amoebot model* [10], programmable matter consists of individual, homogeneous computational elements called *amoebots*. The structure of an amoebot system is represented as a subgraph of an infinite, undirected graph $G = (V, E)$ where V represents all relative positions an amoebot can occupy and E represents all atomic movements an amoebot can make. Each node in V can be occupied by at most one amoebot at a time. Here, we adopt the geometric space variant in which $G = G_\Delta$, the triangular lattice (Figure 1a).

An amoebot has two *shapes*: CONTRACTED, meaning it occupies a single node in V, and EXPANDED, meaning it occupies a pair of adjacent nodes in V (Figure 1b). Each amoebot keeps a collection of ports – one for each edge incident to the node(s) it occupies – that are labeled consecutively according to its own local, persistent *orientation*. All results in this work allow for assorted orientations, meaning amoebots may disagree on both direction (which incident edge points "north") and chirality (clockwise vs. counter-clockwise rotation). Two amoebots occupying adjacent nodes are said to be *neighbors*. Although each amoebot is *anonymous*, lacking a unique identifier, an amoebot can locally identify its neighbors using their port labels. In particular, amoebots A and B connected via ports p_A and p_B know each other's orientations and labels for p_A and p_B.

Each amoebot has memory whose size is a model variant; all results in this work assume constant-size memories. An amoebot's memory consists of two parts: a persistent *public memory* that is only accessible to an amoebot algorithm via communication operations (defined next) and a volatile *private memory* that is directly accessible by amoebot algorithms for temporary variables, computation, etc. *Operations* define the programming interface for amoebot algorithms to communicate and move (see [10] for details):

- The CONNECTED operation tests the presence of neighbors. CONNECTED(p) returns TRUE if and only if there is a neighbor connected via port p.
- The READ and WRITE operations exchange information in public memory. READ(p, x) issues a request to read the value of a variable x in the public memory of the neighbor connected via port p while WRITE(p, x, x_{val}) issues a request to update its value to x_{val}. If $p = \perp$, an amoebot's own public memory is accessed instead of a neighbor's.
- An expanded amoebot can CONTRACT into either node it occupies; a contracted amoebot can EXPAND into an unoccupied adjacent node. Neighboring amoebots can coordinate their movements in a *handover*, which occurs in one of two ways. A contracted amoebot A can PUSH an expanded neighbor B by expanding into a node occupied by B, forcing it to contract. Alternatively, an expanded amoebot B can PULL a contracted neighbor A by contracting, forcing A to expand into the node it is vacating.

Amoebot algorithms are sets of *actions*, each of the form $\langle label \rangle : \langle guard \rangle \to \langle operations \rangle$. An action's *label* specifies its name. Its *guard* is a Boolean predicate determining whether an amoebot A can execute it based on the ports A has connections on – i.e., which nodes adjacent to A are (un)occupied – and information from the public memories of A and its neighbors. An action is *enabled* for an amoebot A if its guard is true for A, and an amoebot is *enabled* if it has at least one enabled action. An action's *operations* specify the finite sequence of operations and computation in private memory to perform if this action is executed.

An amoebot is *active* while executing an action and is *inactive* otherwise. An *adversary* controls the timing of amoebot activations and the resulting action executions, whose *concurrency* and *fairness* are assumption variants. In this work, we consider two concurrency variants: sequential, in which at most one amoebot can be active at a time; and asynchronous, in which any set of amoebots can be simultaneously active. We consider the most general fairness variant: unfair, in which the adversary may activate any enabled amoebot.

An amoebot algorithm's time complexity is evaluated in terms of *rounds* representing the time for the slowest continuously enabled amoebot to execute a single action. Let t_i denote the time at which round $i \in \{0, 1, 2, \ldots\}$ starts, where $t_0 = 0$, and let $\mathcal{E}_i$ denote the set of amoebots that are enabled or already executing an action at time t_i. Round i completes at the earliest time $t_{i+1} > t_i$ by which every amoebot in $\mathcal{E}_i$ either completed an action execution or became disabled at some time in $(t_i, t_{i+1}]$.

2.2 Extensions for Energy Modeling

In addition to the standard model, we introduce new assumptions and terminology specific to modeling energy in amoebot systems. We consider amoebot systems that are finite, initially connected, and contain at least one *source amoebot* with access to an external energy source. Although system connectivity is not generally required by the (canonical) amoebot model, it is necessary for sharing energy from a single source amoebot to the rest of the system via module-to-module power transfer. Each amoebot A has an *energy battery* denoted $A.e_{bat}$ with capacity $\kappa > 0$ representing energy that A can use to perform actions or share with its neighbors (Figure 1c). In this paper, we assume $\kappa = \Theta(1)$ is a fixed integer constant that does not scale with the number of amoebots n, but all results in this paper would hold even if $\kappa = \mathcal{O}(n)$. Source amoebots can harvest energy directly into their batteries while those without access depend on their neighbors to share with them. In either case, we assume an

amoebot transfers at most a single unit of energy per activation.[1] For modeling purposes, we treat $A.e_{bat}$ as a variable stored in the public memory of A. An amoebot A harvesting energy from an external source can be expressed as $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) + 1)$ and likewise an amoebot A transferring energy to a neighbor B connected via a port p is a pair of operations $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) - 1)$ and $\textsc{Write}(p, e_{bat}, \textsc{Read}(p, e_{bat}) + 1)$.

The energy costs for an amoebot algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ are given by a *demand function* $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$; i.e., an amoebot must use $\delta(\alpha_i)$ energy to execute action α_i. Energy is incorporated into actions $\alpha_i \in \mathcal{A}$ by (1) including $A.e_{bat} \geq \delta(\alpha_i)$ in each guard g_i and (2) setting $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) - \delta(\alpha_i))$ as the first operation of ops_i to spend the corresponding amount of energy. An amoebot A is *deficient* w.r.t. an action $\alpha_i \in \mathcal{A}$ if $A.e_{bat} < \delta(\alpha_i)$. An amoebot algorithm $\mathcal{A}$ is *energy-agnostic* if it is not associated with a demand function δ and is *energy-constrained* (w.r.t. δ) otherwise.

The remainder of this paper is dedicated to transforming amoebot algorithms that were designed for the energy-agnostic setting into algorithms with equivalent behavior in the energy-constrained setting w.r.t. any valid demand function under an unfair adversary.

3 A General Framework for Energy-Constrained Algorithms

Amoebot algorithm designers prove the correctness of their algorithms with respect to a *safety* condition (related to the desired system behavior) and a *liveness* condition (ensuring that until this behavior is achieved, some amoebot can make progress towards it). Moving from energy-agnosticism to respecting energy constraints does not affect safety, but may threaten liveness. Some amoebot that was critical to achieving progress in the energy-agnostic setting may now be deficient under the constraints of actions' energy costs, deadlocking the system until it is provided with sufficient energy. Since not all amoebots have access to an external energy source, simply waiting to recharge is not an option. There must be an active strategy for energy distribution embedded in any energy-constrained algorithm.

Instead of placing the burden on algorithm designers to create bespoke implementations of energy distribution for each algorithm, we introduce a general *energy distribution framework*. This framework transforms energy-agnostic algorithms $\mathcal{A}$ that terminate under an unfair adversary and satisfy certain *conventions* into algorithms $\mathcal{A}^\delta$ that are energy-constrained w.r.t. any valid demand function δ and retain their unfair correctness. We give a narrative description and pseudocode for our framework in Section 3.1 and analyze it in Section 3.2.

3.1 The Energy Distribution Framework

Our *energy distribution framework* (Algorithm 1) takes as input any energy-agnostic amoebot algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$ and outputs an energy-constrained algorithm $\mathcal{A}^\delta = \{[\alpha_i^\delta : g_i^\delta \to ops_i^\delta] : i \in \{1, \ldots, m\}\} \cup \{\alpha_{\textsc{EnergyDistribution}}\}$, where actions α_i^δ are energy-constrained versions of the original actions and $\alpha_{\textsc{EnergyDistribution}}$ is a new action that handles energy distribution. Algorithm $\mathcal{A}^\delta$ will achieve the same system behavior as algorithm $\mathcal{A}$ so long as $\mathcal{A}$ satisfies certain conventions:

[1] One could assume that the battery capacity $\kappa > 0$ is any positive real number and that the energy demands are $\delta : \mathcal{A} \to (0, \kappa]$. However, this generality complicates our analysis without meaningfully extending our results, so we make the simplifying assumption that there exists a fundamental unit of energy that divides all action demands $\delta(\alpha_i)$ and the battery capacity κ.

Table 1 Variables used in the Energy Distribution Framework.

Variable	Notation	Domain	Initialization
Forest State	`state`	{SOURCE, IDLE, ACTIVE, ASKING, GROWING, PRUNING}	$\begin{cases}\text{SOURCE} & \text{if source amoebot;}\\ \text{IDLE} & \text{otherwise.}\end{cases}$
Parent Pointer	`parent`	$\{\text{NULL}, 0, \ldots, 9\}$[2]	NULL
Battery Energy	e_{bat}	$\{0, 1, 2, \ldots, \kappa\}$	0

▶ **Definition 1.** *An energy-agnostic amoebot algorithm $\mathcal{A}$ is energy-compatible – i.e., it is compatible with the energy distribution framework – if every (unfair) sequential execution of $\mathcal{A}$ terminates and $\mathcal{A}$ satisfies Conventions 1–3 (defined below).*

Our first two conventions are taken directly from the analogous concurrency control framework for amoebot algorithms [10]. The first convention requires an algorithm's actions to execute successfully in isolation, allowing the framework to ignore invalid actions like attempting to Read on a disconnected port or Expand when already expanded. Formally, we define a *system configuration* as the mapping of amoebots to the node(s) they occupy and the contents of each amoebot's public memory. Throughout the remainder of this paper, we assume configurations are *legal*; i.e., they meet the requirements of the amoebot model.

▶ **Convention 1** (Validity). *All actions α of an amoebot algorithm $\mathcal{A}$ should be valid, i.e., for all (legal) system configurations in which α is enabled for some amoebot A, the execution of α by A should be successful whenever all other amoebots are inactive.*

The second convention defines a common structure for an algorithm's actions by controlling the order and number of their operations, similar to the "look-compute-move" paradigm in the mobile robots literature [17].

▶ **Convention 2** (Phase Structure). *Each action of an amoebot algorithm $\mathcal{A}$ should structure its operations as: (1) a compute phase, during which an amoebot performs a finite amount of computation and a finite sequence of Connected, Read, and Write operations, and (2) a move phase, during which an amoebot performs at most one movement operation decided upon in the compute phase. In particular, no action should use the canonical amoebot model's concurrency control operations, Lock and Unlock.*

Our third and final convention is specific to the energy distribution framework. Recall from Section 2.2 that we consider amoebot systems that are initially connected. This last convention requires an algorithm to maintain system connectivity throughout its execution, ensuring that every amoebot has a path to a source amoebot with access to external energy.

▶ **Convention 3** (Connectivity). *All system configurations reachable by any sequential execution of an amoebot algorithm $\mathcal{A}$ starting in a connected configuration must also be connected.*

Framework Overview. With the conventions defined, we now describe how the energy distribution framework (Algorithm 1) transforms an energy-compatible algorithm $\mathcal{A}$ and a demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$ into an energy-constrained algorithm $\mathcal{A}^\delta$ with

[2] Amoebots maintain one port per incident lattice edge (see Section 2.1), so an expanded amoebot has ten ports despite having a maximum of eight neighbors.

Algorithm 1 Energy Distribution Framework for Amoebot A.

Input: An energy-compatible algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \dots, m\}\}$ and a demand function $\delta : \mathcal{A} \to \{1, 2, \dots, \kappa\}$.

for each action $[\alpha_i : g_i \to ops_i] \in \mathcal{A}$ **do** construct action $\alpha_i^\delta : g_i^\delta \to ops_i^\delta$ as:
  Set $g_i^\delta \leftarrow \big(g_i \wedge (A.e_{bat} \geq \delta(\alpha_i)) \wedge (\forall B \in N(A) \cup \{A\} : B.\texttt{state} \notin \{\text{IDLE}, \text{PRUNING}\})\big)$.
  Set $ops_i^\delta \leftarrow$ "Do:
    $\text{WRITE}(\bot, e_{bat}, \text{READ}(\bot, e_{bat}) - \delta(\alpha_i))$.
    Execute the compute phase of ops_i.
    if the movement phase of ops_i contains a movement operation M_i **then**
      if M_i is $\text{CONTRACT}()$ or $\text{PULL}(p)$ **then**
        $\text{WRITE}(\bot, \texttt{parent}, \text{NULL})$ and $\text{PRUNE}()$.
      else if M_i is $\text{PUSH}(p)$ **then**
        $\text{WRITE}(\bot, \texttt{parent}, \text{NULL})$ and $\text{WRITE}(p, \texttt{parent}, \text{NULL})$.
        $\text{WRITE}(\bot, \texttt{state}, \text{PRUNING})$ and $\text{WRITE}(p, \texttt{state}, \text{PRUNING})$.
      Execute M_i."
Construct $\alpha_{\text{ENERGYDISTRIBUTION}} : g_{\text{ENERGYDISTRIBUTION}} \to ops_{\text{ENERGYDISTRIBUTION}}$ as:
  Set $g_{\text{ENERGYDISTRIBUTION}} \leftarrow \bigvee_{g \in \mathcal{G}}(g)$, where $\mathcal{G} = \{$

$$
\begin{aligned}
g_{\text{GETPRUNED}} &= (A.\texttt{state} = \text{PRUNING}),\\
g_{\text{ASKGROWTH}} &= (A.\texttt{state} = \text{ACTIVE}) \wedge (A \text{ has an IDLE neighbor or ASKING child}),\\
g_{\text{GROWFOREST}} &= (A.\texttt{state} = \text{GROWING}) \vee\\
&\quad \big((A.\texttt{state} = \text{SOURCE}) \wedge (A \text{ has an IDLE neighbor or ASKING child})\big),\\
g_{\text{HARVESTENERGY}} &= (A.\texttt{state} = \text{SOURCE}) \wedge (A.e_{bat} < \kappa),\\
g_{\text{SHAREENERGY}} &= (A.\texttt{state} \notin \{\text{IDLE}, \text{PRUNING}\}) \wedge\\
&\quad (A.e_{bat} \geq 1) \wedge (A \text{ has a child } B : B.e_{bat} < \kappa)\}
\end{aligned}
$$

  Set $ops_{\text{ENERGYDISTRIBUTION}} \leftarrow$ "Do:
    if $g_{\text{GETPRUNED}}$ **then** $\text{PRUNE}()$. ▷ GETPRUNED
    if $g_{\text{ASKGROWTH}}$ **then** $\text{WRITE}(\bot, \texttt{state}, \text{ASKING})$. ▷ ASKGROWTH
    if $g_{\text{GROWFOREST}}$ **then** ▷ GROWFOREST
      for each port p for which $\text{CONNECTED}(p) = \text{TRUE}$ and $\text{READ}(p, \texttt{state}) = \text{IDLE}$ **do**
        $\text{WRITE}(p, \texttt{parent}, p')$, where p' is any port of the neighbor on port p facing A.
        $\text{WRITE}(p, \texttt{state}, \text{ACTIVE})$.
      for each port $p \in \text{CHILDREN}() : (\text{READ}(p, \texttt{state}) = \text{ASKING})$ **do**
        $\text{WRITE}(p, \texttt{state}, \text{GROWING})$.
      if $\text{READ}(\bot, \texttt{state}) = \text{GROWING}$ **then** $\text{WRITE}(\bot, \texttt{state}, \text{ACTIVE})$.
    if $g_{\text{HARVESTENERGY}}$ **then** $\text{WRITE}(\bot, e_{bat}, \text{READ}(\bot, e_{bat}) + 1)$. ▷ HARVESTENERGY
    if $g_{\text{SHAREENERGY}}$ **then** ▷ SHAREENERGY
      Let port $p \in \text{CHILDREN}()$ be one for which $\text{READ}(p, e_{bat}) < \kappa$.
      $\text{WRITE}(\bot, e_{bat}, \text{READ}(\bot, e_{bat}) - 1)$.
      $\text{WRITE}(p, e_{bat}, \text{READ}(p, e_{bat}) + 1)$."
return $\mathcal{A}^\delta = \{[\alpha_i^\delta : g_i^\delta \to ops_i^\delta] : i \in \{1, \dots, m\}\} \cup \{\alpha_{\text{ENERGYDISTRIBUTION}}\}$.

function $\text{CHILDREN}()$
  return $\{\text{ports } p : \text{CONNECTED}(p) \wedge (\text{READ}(p, \texttt{parent}) \text{ points to } A)\}$.
function $\text{PRUNE}()$
  for each port $p \in \text{CHILDREN}()$ **do**
    $\text{WRITE}(p, \texttt{state}, \text{PRUNING})$.
    $\text{WRITE}(p, \texttt{parent}, \text{NULL})$.
  if $\text{READ}(\bot, \texttt{state}) \neq \text{SOURCE}$ **then** $\text{WRITE}(\bot, \texttt{state}, \text{IDLE})$.

"equivalent" behavior (defined formally in Section 3.2). At a high level, $\mathcal{A}^\delta$ works as follows. The amoebot system first self-organizes as a spanning forest $\mathcal{F}$ rooted at source amoebots with access to external energy sources. Energy is harvested by source amoebots and transferred from parents to children in $\mathcal{F}$ as there is need. Amoebots spend energy on enabled actions of algorithm $\mathcal{A}$ until they become deficient, when they will once again need to wait to recharge. This process repeats until termination, which must occur since $\mathcal{A}$ is energy-compatible.

Algorithm $\mathcal{A}^\delta$ comprises two types of actions. First, every action $\alpha_i \in \mathcal{A}$ is transformed into an energy-constrained version $\alpha_i^\delta \in \mathcal{A}^\delta$ (Algorithm 1, Lines 1–12). By including $A.e_{bat} \geq \delta(\alpha_i)$ in its guard g_i^δ and spending $\delta(\alpha_i)$ energy at the start of its operations ops_i^δ, the transformed action α_i^δ is only executed if there is sufficient energy to do so and any such execution spends the corresponding energy. The guard g_i^δ also ensures any amoebot executing an α_i^δ action and all of its neighbors are part of the forest structure $\mathcal{F}$.

Second, there is a singular $\alpha_{\text{EnergyDistribution}}$ action that defines how amoebots self-organize as a spanning forest and distribute energy throughout the system (Algorithm 1, Lines 13–30). Its operations are organized into five blocks – GetPruned, AskGrowth, GrowForest, HarvestEnergy, and ShareEnergy– each of which has a corresponding logical predicate in the set $\mathcal{G}$. These predicates appear in the guard $\bigvee_{g\in\mathcal{G}}(g)$, which ensures that $\alpha_{\text{EnergyDistribution}}$ is only enabled when its execution would progress towards distributing energy to deficient amoebots. The latter is critical for proving that $\mathcal{A}^\delta$ achieves energy distribution even under an unfair adversary, which we show in Section 3.2. The remainder of this section details the five blocks; their local variables are summarized in Table 1.

Forming and Maintaining a Spanning Forest. Recall from Section 2.2 that we consider amoebot systems that are initially connected and contain at least one source amoebot with access to an external energy source. The GetPruned, AskGrowth, and GrowForest blocks (Algorithm 1, Lines 17–25) continuously organize the amoebot system as a spanning forest $\mathcal{F}$ of trees rooted at the source amoebot(s). These trees act as an acyclic resource distribution network for energy transfers, which is important for avoiding non-termination under an unfair adversary.

The well-established *spanning forest primitive* [9] and the recent *feather tree formation* algorithm [25] are both guaranteed to organize an amoebot system as a spanning forest $\mathcal{F}$ under an unfair sequential adversary, assuming no parent–child relationship in $\mathcal{F}$ is ever disrupted after it is formed. However, many amoebot algorithms $\mathcal{A}$ – and by extension, the actions α_i^δ of algorithms $\mathcal{A}^\delta$ – cause amoebots to move, partitioning $\mathcal{F}$ into "unstable" trees whose connections to source amoebots have been disrupted and "stable" trees that remain rooted at sources. This necessitates a protocol for dynamically repairing $\mathcal{F}$ as amoebots move. To this end, the earlier Forest-Prune-Repair algorithm [11] was designed to "prune" unstable trees, allowing their amoebots to rejoin stable trees. Unfortunately, Forest-Prune-Repair requires fairness for termination, which we do not have here. In the following, we describe a new algorithm that dynamically maintains $\mathcal{F}$ under an unfair sequential adversary.

Each amoebot has a `state` variable that is initialized to SOURCE for source amoebots and IDLE for all others. Additionally, each amoebot has a `parent` pointer indicating the port incident to their parent in the forest $\mathcal{F}$; these pointers are initially set to NULL. A source amoebot adopts its IDLE neighbors into its tree by making them ACTIVE and setting their `parent` pointers to itself (GrowForest, Algorithm 1, Lines 19–22). ACTIVE amoebots, however, must ask the source amoebot at the root of their tree for permission before adopting their IDLE neighbors (AskGrowth, Algorithm 1, Line 18). Although indirect, this ensures that IDLE amoebots only join trees that are (or were recently) stable, stopping the unfair

adversary from creating non-terminating executions (see Lemma 4). Specifically, an ACTIVE amoebot with an IDLE neighbor becomes ASKING. Any ACTIVE amoebot with an ASKING child also becomes ASKING, propagating this "asking signal" towards the tree's source amoebot. When the source amoebot receives this asking signal, it updates all its ASKING children to GROWING, granting them permission to grow the tree. A GROWING amoebot adopts its IDLE neighbors as ACTIVE children, updates its ASKING children to GROWING, and resets its `state` to ACTIVE. This process repeats until no IDLE amoebots remain.

If an amoebot's movement during an α_i^δ execution would disrupt $\mathcal{F}$, it initiates a pruning process to dissolve disrupted subtrees. Amoebots performing CONTRACT or PULL movements must prune immediately since their movement may disconnect them from their neighbors; PUSH movements instead make the two involved amoebots PRUNING, which will cause them to prune during their next action. When an amoebot prunes, it makes its children PRUNING and resets both its own and its children's `parent` pointers, severing them from their tree (Algorithm 1, Lines 8 and 35–37). If it is not a source, it also becomes IDLE (Algorithm 1, Line 38). The GETPRUNED block ensures that any PRUNING amoebot does the same, dissolving the unstable tree (Algorithm 1, Line 17). These newly IDLE amoebots are then collected into stable trees by the ASKGROWTH and GROWFOREST blocks as described above.

Sharing Energy. The HARVESTENERGY and SHAREENERGY blocks (Algorithm 1, Lines 26–30) define how source amoebots harvest energy from external energy sources and how all non-IDLE, non-PRUNING amoebots transfer energy to their neighbors, respectively. If its battery is not already full, a source amoebot harvests a unit of energy from its external energy source into its own battery. Any non-IDLE, non-PRUNING amoebot with at least one unit of energy to share and a child whose battery is not full will then transfer a unit of energy from its own battery to that of its child.

3.2 Analysis

In this section, we sketch the results building to the following theorem. Informally, it states that an energy-constrained algorithm $\mathcal{A}^\delta$ produced by the energy distribution framework (1) only yields system outcomes that could have been achieved by the original energy-agnostic algorithm $\mathcal{A}$, provided $\mathcal{A}$ is energy-compatible, and (2) incurs an $\mathcal{O}(n^2)$ runtime overhead.

▶ **Theorem 2.** *Consider any energy-compatible amoebot algorithm $\mathcal{A}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$, and let $\mathcal{A}^\delta$ be the algorithm produced from $\mathcal{A}$ and δ by the energy distribution framework (Algorithm 1). Let C_0 be any (legal) connected initial configuration for $\mathcal{A}$ and let C_0^δ be its extension for $\mathcal{A}^\delta$ that designates at least one source amoebot and adds the energy distribution variables with their initial values (Table 1) to all amoebots. Then for any configuration C^δ in which an unfair sequential execution of $\mathcal{A}^\delta$ starting in C_0^δ terminates, there exists an unfair sequential execution of $\mathcal{A}$ starting in C_0 that terminates in a configuration C that is identical to C^δ modulo the energy distribution variables. Moreover, if all unfair sequential executions of $\mathcal{A}$ on n amoebots terminate after at most $T_\mathcal{A}(n)$ action executions, then any unfair sequential execution of $\mathcal{A}^\delta$ on n amoebots terminates in $\mathcal{O}(n^2 T_\mathcal{A}(n))$ rounds.*

Due to space constraints, we highlight only the most important supporting results of this analysis. All omitted lemmas, invariants, and proofs can be found in the full version of this paper (see link in title page).

▶ **Lemma 3.** *Consider any sequential execution $\mathcal{S}^\delta$ of $\mathcal{A}^\delta$ starting in initial configuration C_0^δ and let $\mathcal{S}_\alpha^\delta$ denote its subsequence of α_i^δ action executions. Then the corresponding sequence $\mathcal{S}_\alpha$ of α_i executions is a valid sequential execution of $\mathcal{A}$ starting in initial configuration C_0.*

This lemma implies that any sequential execution $\mathcal{S}^\delta$ of $\mathcal{A}^\delta$ contains a finite number of α_i^δ executions, since the corresponding sequence of α_i executions forms a possible sequential execution of $\mathcal{A}$, which must terminate because $\mathcal{A}$ is energy-compatible. It remains to analyze the *energy runs* in $\mathcal{S}^\delta$, i.e., the maximal sequences of consecutive $\alpha_{\text{EnergyDistribution}}$ executions delineated by α_i^δ executions. Formally, an execution of $\alpha_{\text{EnergyDistribution}}$ by an amoebot A is *g-supported* if predicate $g \in \mathcal{G}$ is satisfied when A is activated and executes $\alpha_{\text{EnergyDistribution}}$. We argue that any predicate $g \in \mathcal{G}$ can support at most a finite number of executions per energy run, implying that all energy runs, and thus all sequential executions of $\mathcal{A}^\delta$, are finite:

▶ **Lemma 4.** *Any energy run of $\mathcal{S}^\delta$ contains at most a finite number of g-supported $\alpha_{\text{EnergyDistribution}}$ executions, for any $g \in \mathcal{G}$.*

Let C^δ be the terminating configuration of $\mathcal{S}^\delta$. We must show that there exists a sequential execution of $\mathcal{A}$ starting in C_0 that terminates in the configuration C obtained from C^δ by removing the energy distribution variables. An obvious candidate is the sequence $\mathcal{S}_\alpha$ of α_i executions corresponding to the α_i^δ executions in $\mathcal{S}^\delta$. Lemma 3 already implies that $\mathcal{S}_\alpha$ reaches C, and a careful argument involving the guard of $\alpha_{\text{EnergyDistribution}}$ shows that it must also terminate there. The remainder of the analysis characterizes the time required for an *uninterrupted* energy run – i.e., one that is not ended early by an α_i^δ execution, which only helps the overall progress argument – to collect all amoebots into stable trees rooted at source amoebots and, once this is achieved, to fully recharge all amoebots' batteries.

▶ **Lemma 5.** *After at most $\mathcal{O}(n^2)$ rounds of any uninterrupted energy run of $\mathcal{S}^\delta$, all n amoebots belong to stable trees.*

▶ **Lemma 6.** *After at most $\mathcal{O}(n)$ rounds of any uninterrupted, stabilized energy run of $\mathcal{S}^\delta$, all n amoebots have full batteries.*

These lemmas imply that every energy run terminates in at most $\mathcal{O}(n^2)$ rounds. The theorem supposes that any sequential execution of $\mathcal{A}$ terminates in $T_\mathcal{A}(n)$ action executions, so we know by Lemma 3 that any sequential execution of $\mathcal{A}^\delta$ contains at most $T_\mathcal{A}(n) + 1$ energy runs. Combining these facts yields the $\mathcal{O}(n^2 T_\mathcal{A}(n))$ runtime bound for $\mathcal{A}^\delta$.

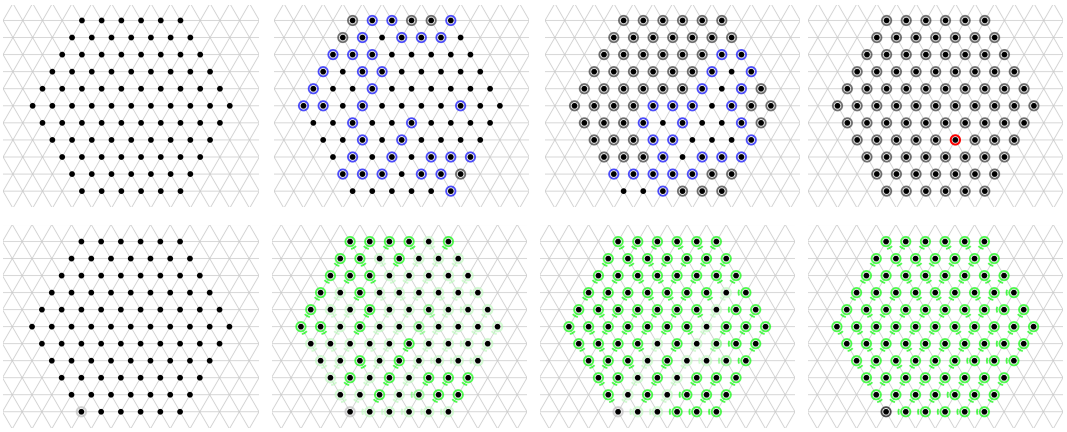
4 Energy-Constrained Leader Election and Shape Formation

With the energy distribution framework defined and its properties analyzed, we now apply it to existing energy-agnostic algorithms for leader election and shape formation and show simulations of their energy-constrained counterparts. We first make a straightforward observation about *stationary* amoebot algorithms, i.e., those in which amoebots do not move. These include simple primitives like spanning forest formation [9] and binary counters [7, 33] as well as the majority of existing algorithms for leader election [3, 5, 8, 14, 15, 18, 19]. It is easily seen that an algorithm that never moves cannot disconnect an initially connected system, and its actions never involve a "move phase". Thus,

▶ **Observation 7.** *All stationary amoebot algorithms satisfy Convention 3, and those that do not use Lock or Unlock operations also satisfy Convention 2.*

Observation 7 immediately implies the following about stationary algorithms' compatibility with the energy distribution framework.

▶ **Corollary 8.** *Any stationary amoebot algorithm that terminates under every (unfair) sequential execution, comprises only valid actions (i.e., those whose executions always succeed in isolation), and does not use Lock or Unlock operations is energy-compatible.*



(a) $t = 0$ rounds. **(b)** $t = 100$. **(c)** $t = 250$. **(d)** $t = 350$.

Figure 2 *Simulating Leader-Election-by-Erosion$^\delta$.* A simulation of Leader-Election-by-Erosion$^\delta$ on $n = 91$ amoebots with one source amoebot, capacity $\kappa = 10$, and demand $\delta(\alpha) = 5$ for all actions α. Both rows show the same simulation. Top: For Leader-Election-by-Erosion, amoebots are initially "null candidates" (no color) and eventually declare candidacy (blue); candidates then either erode (dark gray) or become the unique leader (red). Bottom: For energy distribution, color opacity indicates energy levels. All amoebots are initially IDLE (no color) except the source (gray/black); amoebots eventually join the forest $\mathcal{F}$ (green) and distribute energy.

One such algorithm is Leader-Election-by-Erosion, a deterministic leader election algorithm for hole-free, connected amoebot systems introduced by Di Luna et al. [15] and extended to the canonical amoebot model and three-dimensional space by Briones et al. [5]. All amoebots first become leader candidates. When activated, a candidate uses certain rules regarding the number and relative positions of its neighbors to decide whether to "erode", revoking its candidacy without disconnecting or introducing a hole into the remaining set of candidates. The last remaining candidate is necessarily unique and thus declares itself the leader.

▶ **Lemma 9.** *Leader-Election-by-Erosion is energy-compatible.*

Proof. Leader-Election-by-Erosion is clearly stationary – no movement is involved in checking neighbors' positions or revoking candidacy – so it suffices to check the conditions of Corollary 8. Briones et al. [5] have already shown that any unfair sequential execution of this algorithm elects a leader – and thus terminates – in $\mathcal{O}(n)$ rounds. This correctness analysis also confirms that no actions of Leader-Election-by-Erosion are invalid; otherwise, some action executions would fail. Finally, it is easy to verify from the algorithm's pseudocode in [5] that LOCK and UNLOCK are not used, so we are done. ◀

Combining this lemma, the energy distribution framework's guarantees (Theorem 2), and Leader-Election-by-Erosion's correctness and runtime guarantees (Theorem 6.3 of [5]) immediately implies the following theorem.

▶ **Theorem 10.** *For any demand function δ : Leader-Election-by-Erosion $\to \{1, 2, \ldots, \kappa\}$, the algorithm Leader-Election-by-Erosion$^\delta$ produced by the energy distribution framework deterministically solves the leader election problem for hole-free, connected systems of n amoebots in $\mathcal{O}(n^3)$ rounds assuming geometric space, assorted orientations, constant-size memory, and an unfair sequential adversary.*

A simulation of Leader-Election-by-Erosion$^\delta$ successfully electing a unique leader under energy constraints is shown in Figure 2. As the proof of Lemma 9 shows, Corollary 8 sets a very low bar for proving stationary algorithms are energy-compatible. Almost all existing

amoebot algorithms are designed to terminate after achieving a desired system behavior, and this property is typically proven as part of their correctness analyses. Invalid actions are avoided, as their executions would always fail.[3] Finally, no existing algorithms use the concurrency control operations Lock and Unlock directly; these are typically reserved for use by the "concurrency control framework" [10] discussed in the next section. The only remaining obstacle is that many existing stationary algorithms predate the canonical amoebot model and have not yet been reformulated in guarded action semantics or analyzed under an unfair adversary. Supposing this obstacle can be overcome without significantly affecting the algorithms' previously proven guarantees, the above discussion shows it is likely that most – if not all – existing stationary amoebot algorithms are energy-compatible.

What about non-stationary amoebot algorithms whose movements make satisfying the phase structure and connectivity conventions (Conventions 2 and 3) non-trivial? Here our example is the Hexagon-Formation algorithm for basic shape formation, originally introduced by Derakhshandeh et al. [13] and carefully reformulated and analyzed under the canonical amoebot model by Daymude et al. [10]. The basic idea of this algorithm is to form a hexagon – or as close to one as is possible with the number of amoebots in the system – by extending a spiral that begins at a (pre-defined or elected) seed amoebot. Thanks to the analysis in [10], it is easy to show Hexagon-Formation is compatible with the energy distribution framework.

▶ **Lemma 11.** *Hexagon-Formation is energy-compatible.*

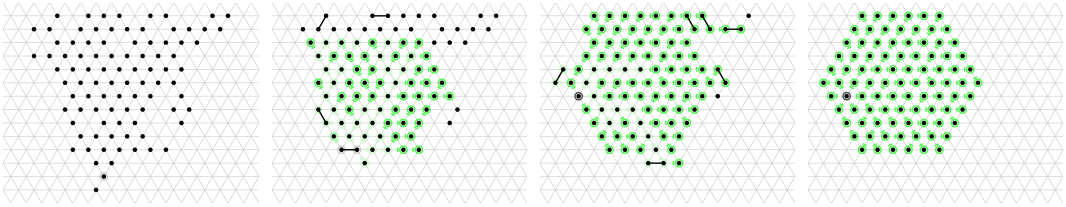
Proof. Every sequential execution of Hexagon-Formation must terminate since Lemma 7 of [10] guarantees that any execution of this algorithm – sequential or concurrent – terminates with the amoebot system forming a hexagon. Theorem 10 of [10] guarantees that Hexagon-Formation satisfies the validity and phase structure conventions (Conventions 1 and 2), as these were the two conventions borrowed directly from that paper's concurrency control framework. Finally, Hexagon-Formation is guaranteed to maintain the connectivity of an initially connected system configuration by Lemma 3 of [10], satisfying Convention 3. ◀

Combining this lemma, the energy distribution framework's guarantees (Theorem 2), Hexagon-Formation's correctness guarantees (Theorem 8 of [10]), and Hexagon-Formation's $\Theta(n^2)$ worst-case work bound [13], we have:

▶ **Theorem 12.** *For any demand function δ : Hexagon-Formation $\to \{1, 2, \ldots, \kappa\}$, the algorithm Hexagon-Formation$^\delta$ produced by the energy distribution framework deterministically solves the hexagon formation problem for connected systems of n amoebots in $\mathcal{O}(n^4)$ rounds assuming geometric space, assorted orientations, constant-size memory, and an unfair sequential adversary.*

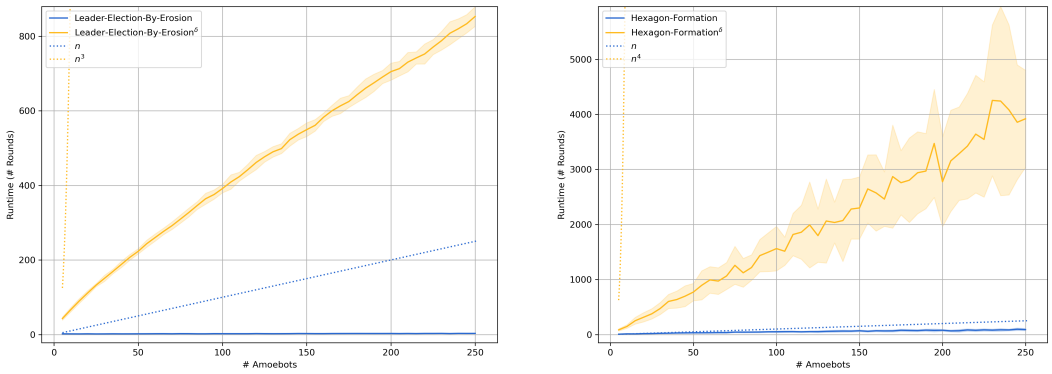
Figure 3 depicts a simulation of Hexagon-Formation$^\delta$ forming a hexagon under energy constraints. We emphasize that Leader-Election-by-Erosion and Hexagon-Formation are not cherry-picked examples with particularly straightforward proofs of energy-compatibility. On the contrary, we expect that like our two examples, many algorithms already have the ingredients of energy-compatibility proven in their existing correctness analyses.

[3] The canonical amoebot model introduced error handling for amoebot algorithm design to deal with operation executions that fail due to concurrency (see Section 2.2 of [10]). Although error handling could be used to deal with failed executions of invalid actions, no existing amoebot algorithms have taken such a convoluted approach to designing functional algorithms.



(a) $t = 0$ rounds. **(b)** $t = 400$. **(c)** $t = 900$. **(d)** $t = 1200$.

Figure 3 *Simulating Hexagon-Formation$^\delta$.* A simulation of Hexagon-Formation$^\delta$ on $n = 91$ amoebots with one source amoebot, capacity $\kappa = 10$, and demand $\delta(\alpha) = 5$ for all actions α. States from Hexagon-Formation are not visualized. For energy distribution, color opacity indicates energy levels. All amoebots are initially IDLE (no color) except the source (gray/black); amoebots eventually join the forest $\mathcal{F}$ (green) and distribute energy.



(a) Leader-Election-by-Erosion. **(b)** Hexagon-Formation.

Figure 4 *Runtime Comparisons.* The energy-constrained (a) Leader-Election-by-Erosion$^\delta$ and (b) Hexagon-Formation$^\delta$ algorithms' runtimes (yellow) and their energy-agnostic counterparts (blue) in terms of sequential rounds. Each algorithm was simulated in 25 independent trials per system size $n \in \{5, 10, \ldots, 250\}$; average runtimes are shown as solid lines and one standard deviation is shown as an error tube. Relevant asymptotic runtime bounds are shown as dotted lines: the energy-agnostic algorithms both terminate in linear rounds (blue) and the energy-constrained algorithms' bounds are given by Theorems 10 and 12 (yellow).

We validate the runtime bounds for Leader-Election-by-Erosion$^\delta$ and Hexagon-Formation$^\delta$ given in Theorems 10 and 12, respectively, by simulating these algorithms and their energy-agnostic counterparts for a range of system sizes n. Figure 4 reports their empirical runtimes. Both energy-constrained algorithms well outperform their theoretical bounds, with Leader-Election-by-Erosion$^\delta$ achieving a near-linear runtime and Hexagon-Formation$^\delta$ remaining sub-quadratic. This suggests that our overhead bound can be optimized further or describes only some pessimistic worst-case scenarios.

5 Asynchronous Energy-Constrained Algorithms

Our energy distribution results thus far consider sequential concurrency, in which at most one amoebot can be active at a time (Section 2.1). This section details a useful extension of these results to *asynchronous concurrency*, in which arbitrary amoebots can be simultaneously active and their action executions can overlap arbitrarily in time.

There are many hazards of asynchrony that complicate amoebot algorithm design, with concurrent movements and memory updates potentially causing operations to fail or action executions to exhibit unintended behaviors. To reduce this complexity, one can use the *concurrency control framework* for amoebot algorithms that – analogous to our own energy distribution framework for energy-agnostic/constrained algorithms – transforms any algorithm $\mathcal{A}$ that terminates under every (unfair) sequential execution and satisfies certain conventions into an algorithm $\mathcal{A}'$ that achieves equivalent behavior under any asynchronous execution [10]. Formally, an amoebot algorithm $\mathcal{A}$ is *concurrency-compatible* if every (unfair) sequential execution of $\mathcal{A}$ terminates and it satisfies the validity, phase structure, and expansion-robustness conventions. The first two conventions are identical to Conventions 1 and 2 of the energy distribution framework. The third convention, *expansion-robustness*, requires actions to be resilient to concurrent expansions into their neighborhood.

We originally aimed to prove that the energy distribution framework preserves any input algorithm's concurrency-compatibility – i.e., if an algorithm $\mathcal{A}$ is concurrency-compatible, then so is $\mathcal{A}^\delta$ – and thus the two frameworks can be composed to obtain energy-constrained, asynchronous versions of all energy-compatible, concurrency-compatible algorithms. But as will become clearer after we formally define expansion-robustness (Definition 13), knowing that $\mathcal{A}$ is expansion-robust is seemingly insufficient for proving that $\mathcal{A}^\delta$ is also expansion-robust: the former only describes terminating configurations for $\mathcal{A}$ while the latter requires analyzing possible amoebot movements in all intermediate configurations reached by $\mathcal{A}^\delta$. Instead, we focus on a special case of expansion-robustness called *expansion-correspondence* (Definition 14) that we can prove is preserved by the energy distribution framework (Lemma 15). Although this restriction may appear limiting, the only algorithm known to be non-trivially expansion-robust (Hexagon-Formation of [10]) was proven to be expansion-robust via expansion-correspondence. Thus, until an algorithm is discovered to be expansion-robust but not expansion-corresponding, our present focus covers all known concurrency-compatible algorithms.

Formally, let $\mathcal{A}$ be any amoebot algorithm satisfying Conventions 1 and 2 and consider its expansion-robust variant $\mathcal{A}^E$ defined as follows. Each amoebot A executing $\mathcal{A}^E$ additionally stores in public memory an *expand flag* $A.\texttt{flag}_p$ for each of its ports p that is initially FALSE, becomes TRUE whenever A expands to reveal a new port p, and is reset to FALSE whenever A or one of its neighbors executes a later action. These expand flags communicate when an amoebot has newly expanded into another amoebot's neighborhood. Each action $\alpha_i : g_i \to ops_i$ in $\mathcal{A}$ becomes an action $\alpha_i^E : g_i^E \to ops_i^E$ in $\mathcal{A}^E$ (see Algorithm 2 in Appendix A for details). The main difference is that while an amoebot A executes actions with respect to its full neighborhood $N(A)$ in $\mathcal{A}$, it does so only with respect to its *established neighborhood* $N^E(A) = \{B \in N(A) : \exists \text{ port } p \text{ of } B \text{ connected to } A \text{ s.t. } B.\texttt{flag}_p = \text{FALSE}\}$ in $\mathcal{A}^E$, effectively ignoring its newly expanded neighbors until its next action execution.

▶ **Definition 13.** *An amoebot algorithm $\mathcal{A}$ is expansion-robust if for any (legal) initial system configuration C_0 of $\mathcal{A}$, the following conditions hold:*

1. *If all sequential executions of $\mathcal{A}$ starting in C_0 terminate, all sequential executions of $\mathcal{A}^E$ starting in C_0^E (i.e., C_0 with all FALSE expand flags) also terminate.*
2. *If a sequential execution of $\mathcal{A}^E$ starting in C_0^E terminates in a configuration C^E, some sequential execution of $\mathcal{A}$ starting in C_0 terminates in C (i.e., C^E without expand flags).*

As alluded to earlier, expansion-robustness only guarantees that sequential executions of $\mathcal{A}^E$ terminate and do so in a configuration that is reachable by a sequential execution of $\mathcal{A}$. This appears to be insufficient to prove $\mathcal{A}^\delta$ is expansion-robust. We instead focus on the following special case of expansion-robustness.

▶ **Definition 14.** *An amoebot algorithm $\mathcal{A}$ is expansion-corresponding if for any (legal) initial system configuration C_0 of $\mathcal{A}$, the following conditions hold:*

1. *If an action $\alpha^E_{i\neq 0} \in \mathcal{A}^E$ is enabled for some amoebot A w.r.t. $N^E(A)$, then action $\alpha_i \in \mathcal{A}$ is enabled for A w.r.t. $N(A)$.*
2. *The executions of $\alpha^E_{i\neq 0}$ w.r.t. $N^E(A)$ and α_i w.r.t. $N(A)$ by an amoebot A are identical, except the handling of expand flags.*

The main lemma of this section proves that the energy distribution framework preserves expansion-correspondence. Its proof and supporting results can be found in Appendix A.

▶ **Lemma 15.** *For any energy-compatible, expansion-corresponding algorithm $\mathcal{A}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$, the algorithm $\mathcal{A}^\delta$ produced from $\mathcal{A}$ and δ by the energy distribution framework is concurrency-compatible.*

Lemma 15 shows that the energy distribution and concurrency control frameworks can be composed to obtain the benefits of both. Specifically, an amoebot algorithm designer should first design their algorithm without energy constraints and perform the usual safety and liveness analyses with respect to an unfair sequential adversary. If the algorithm always terminates, then they need only prove their algorithm satisfies the validity, phase structure, and connectivity conventions and argue that their algorithm is expansion-corresponding to automatically obtain an energy-constrained, asynchronous version of their algorithm with equivalent behavior, courtesy of the two frameworks. The following theorem states this result formally by combining the energy distribution framework's guarantees (Theorem 2), the concurrency control framework's guarantees (Theorem 11 of [10]), and Lemma 15. Note that because the runtime overhead of the concurrency control framework is not known, this theorem does not give any overhead bounds.

▶ **Theorem 16.** *Consider any energy-compatible, expansion-corresponding amoebot algorithm $\mathcal{A}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$. Let $\mathcal{A}^\delta$ be the algorithm produced from $\mathcal{A}$ and δ by the energy distribution framework (Algorithm 1) and let $(\mathcal{A}^\delta)'$ be the algorithm produced from $\mathcal{A}^\delta$ by the concurrency control framework (Algorithm 4 of [10]). Let C_0 be any (legal) connected initial configuration for $\mathcal{A}$ and let $(C_0^\delta)'$ be its extension for $(\mathcal{A}^\delta)'$ that designates at least one source amoebot and adds the energy distribution and concurrency control variables with their initial values (Table 1 and* `act` *and* `awaken` *of [10]) to all amoebots. Then every asynchronous execution of $(\mathcal{A}^\delta)'$ starting in $(C_0^\delta)'$ terminates. Moreover, if $(C^\delta)'$ is the final configuration of some asynchronous execution of $(\mathcal{A}^\delta)'$ starting in $(C_0^\delta)'$, then there exists a sequential execution of $\mathcal{A}$ starting in C_0 that terminates in a configuration C that is identical to $(C^\delta)'$ modulo the energy distribution and concurrency control variables.*

We conclude this section by applying Theorem 16 to the Leader-Election-by-Erosion and Hexagon-Formation algorithms from Section 4. Those algorithms were shown to be energy-compatible in Lemmas 9 and 11 and expansion-corresponding in Lemma 7.1 of [5] and Theorem 10 of [10], respectively. Therefore,

▶ **Corollary 17.** *There exist energy-constrained amoebot algorithms that deterministically solve the leader election problem (for hole-free, connected systems) and the hexagon formation problem (for connected systems) assuming geometric space, assorted orientations, constant-size memory, and an unfair asynchronous adversary – the most general of all adversaries.*

6 Conclusion

In this work, we introduced the energy distribution framework for amoebot algorithms which transforms any energy-agnostic algorithm into an energy-constrained one with equivalent behavior, provided the original algorithm terminates under an unfair sequential adversary, maintains system connectivity, and follows some basic structural conventions (Theorem 2). We then proved that both the Leader-Election-by-Erosion and Hexagon-Formation algorithms are energy-compatible (Theorems 10 and 12). Perhaps surprisingly, these proofs were not difficult. The algorithms' existing correctness and runtime analyses under an unfair sequential adversary provided nearly all that was needed for energy-compatibility, and we expect this would be true for other algorithms as well. Finally, we proved that if an energy-compatible algorithm is also expansion-corresponding, then its energy-constrained counterpart produced by our framework can be extended to asynchronous concurrency using the concurrency control framework for amoebot algorithms (Theorem 16).

The energy-constrained algorithms produced by our framework have an $\mathcal{O}(n^2)$ round runtime overhead, though our simulations of Leader-Election-by-Erosion$^\delta$ and Hexagon-Formation$^\delta$ suggest that the overhead is much lower in practice. Comparing Lemmas 5 and 6 reveals the spanning forest maintenance algorithm as the performance bottleneck, which uses $\mathcal{O}(n^2)$ rounds in the worst case to prune and rebuild a forest of stable trees. In particular, amoebots getting permission from their (source) root before adopting children is critical for avoiding non-termination under an unfair adversary (Lemma 4), but requires a number of rounds that is linear in the depth of the tree. Improving this bound either requires a new approach to acyclic resource distribution or an optimization of stable tree membership detection. A shortest-path tree – i.e., one that maintains equality between the in-tree and in-system distances from any amoebot to its root – would bound the depth of any tree by the diameter D of the system. This would reduce the overall overhead to $\mathcal{O}(nD)$ rounds, which is still $\mathcal{O}(n^2)$ in the worst case (e.g., a line) but could achieve up to $\mathcal{O}(n^{3/2})$ in the best case (e.g., a regular hexagon). However, the recent feather tree algorithm [25] for forming shortest-path forests in amoebot systems only works in stationary systems. Achieving an algorithm for shortest-path forest maintenance – not just formation – would both improve our present overhead bound and be an interesting contribution in its own right.

References

1 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in Networks of Passively Mobile Finite-State Sensors. *Distributed Computing*, 18(4):235–253, 2006. doi:10.1007/S00446-005-0138-3.

2 Palina Bartashevich, Doreen Koerte, and Sanaz Mostaghim. Energy-Saving Decision Making for Aerial Swarms: PSO-Based Navigation in Vector Fields. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2017. doi:10.1109/SSCI.2017.8285178.

3 Rida A. Bazzi and Joseph L. Briones. Stationary and Deterministic Leader Election in Self-Organizing Particle Systems. In *Stabilization, Safety, and Security of Distributed Systems*, volume 11914 of *Lecture Notes in Computer Science*, pages 22–37, 2019. doi:10.1007/978-3-030-34992-9_3.

4 Douglas Blackiston, Emma Lederer, Sam Kriegman, Simon Garnier, Joshua Bongard, and Michael Levin. A Cellular Platform for the Development of Synthetic Living Machines. *Science Robotics*, 6(52):eabf1571, 2021. doi:10.1126/scirobotics.abf1571.

5 Joseph L. Briones, Tishya Chhabra, Joshua J. Daymude, and Andréa W. Richa. Invited Paper: Asynchronous Deterministic Leader Election in Three-Dimensional Programmable Matter. In *Proceedings of the 24th International Conference on Distributed Computing and Networking*, pages 38–47, 2023. doi:10.1145/3571306.3571389.

6 Jason D. Campbell, Padmanabhan Pillai, and Seth Copen Goldstein. The Robot Is the Tether: Active, Adaptive Power Routing for Modular Robots with Unary Inter-Robot Connectors. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4108–4115, 2005. doi:10.1109/IROS.2005.1545426.

7 Joshua J. Daymude, Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Christian Scheideler, and Andréa W. Richa. Convex Hull Formation for Programmable Matter. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, pages 2:1–2:10, 2020. doi:10.1145/3369740.3372916.

8 Joshua J. Daymude, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Improved Leader Election for Self-Organizing Programmable Matter. In *Algorithms for Sensor Systems*, volume 10718 of *Lecture Notes in Computer Science*, pages 127–140, 2017. doi:10.1007/978-3-319-72751-6_10.

9 Joshua J. Daymude, Kristian Hinnenthal, Andréa W. Richa, and Christian Scheideler. Computing by Programmable Particles. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities*, volume 11340 of *Lecture Notes in Computer Science*, pages 615–681. Springer, Cham, 2019. doi:10.1007/978-3-030-11072-7_22.

10 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The Canonical Amoebot Model: Algorithms and Concurrency Control. *Distributed Computing*, 2023. doi:10.1007/s00446-023-00443-3.

11 Joshua J. Daymude, Andréa W. Richa, and Jamison W. Weber. Bio-Inspired Energy Distribution for Programmable Matter. In *International Conference on Distributed Computing and Networking 2021*, pages 86–95, 2021. doi:10.1145/3427796.3427835.

12 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Amoebot – A New Model for Programmable Matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 220–222, 2014. doi:10.1145/2612669.2612712.

13 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. An Algorithmic Framework for Shape Formation Problems in Self-Organizing Particle Systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, pages 21:1–21:2, 2015. doi:10.1145/2800795.2800829.

14 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W. Richa, and Christian Scheideler. Leader Election and Shape Formation with Self-Organizing Programmable Matter. In Andrew Phillips and Peng Yin, editors, *DNA Computing and Molecular Programming*, volume 9211 of *Lecture Notes in Computer Science*, pages 117–132, 2015. doi:10.1007/978-3-319-21999-8_8.

15 Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape Formation by Programmable Particles. *Distributed Computing*, 33(1):69–101, 2020. doi:10.1007/S00446-019-00350-6.

16 Shlomi Dolev, Sergey Frenkel, Michael Rosenblit, Ram Prasadh Narayanan, and K. Muni Venkateswarlu. In-Vivo Energy Harvesting Nano Robots. In *2016 IEEE International Conference on the Science of Electrical Engineering (ICSEE)*, pages 1–5, 2016. doi:10.1109/ICSEE.2016.7806107.

17 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.

18 Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Distributed Leader Election and Computation of Local Identifiers for Programmable Matter. In Seth Gilbert, Danny Hughes, and Bhaskar Krishnamachari, editors, *Algorithms for Sensor Systems*, volume 11410 of *Lecture Notes in Computer Science*, pages 159–179, 2019. doi:10.1007/978-3-030-14094-6_11.

19 Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Leader Election and Local Identifiers for Three-dimensional Programmable Matter. *Concurrency and Computation: Practice and Experience*, 34(7):e6067, 2022. doi:10.1002/CPE.6067.

20 Kyle Gilpin, Ara Knaian, and Daniela Rus. Robot Pebbles: One Centimeter Modules for Programmable Matter through Self-Disassembly. In *2010 IEEE International Conference on Robotics and Automation*, pages 2485–2492, 2010. doi:10.1109/ROBOT.2010.5509817.

21 Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Fabian Kuhn, Dorian Rudolph, Christian Scheideler, and Thim Strothmann. Forming Tile Shapes with Simple Robots. *Natural Computing*, 19(2):375–390, 2020. doi:10.1007/S11047-019-09774-2.

22 Seth Copen Goldstein, Jason D. Campbell, and Todd C. Mowry. Programmable Matter. *Computer*, 38(6):99–101, 2005. doi:10.1109/MC.2005.198.

23 Seth Copen Goldstein, Todd C. Mowry, Jason D. Campbell, Michael P. Ashley-Rollman, Michael De Rosa, Stanislav Funiak, James F. Hoburg, Mustafa E. Karagozler, Brian Kirby, Peter Lee, Padmanabhan Pillai, J. Robert Reid, Daniel D. Stancil, and Michael P. Weller. Beyond Audio and Video: Using Claytronics to Enable Pario. *AI Magazine*, 30(2):29–45, 2009. doi:10.1609/AIMAG.V30I2.2241.

24 Serge Kernbach, editor. *Handbook of Collective Robotics: Fundamentals and Challenges*. Jenny Stanford Publishing, New York, NY, USA, 2013. doi:10.1201/b14908.

25 Irina Kostitsyna, Tom Peters, and Bettina Speckmann. Brief Announcement: An Effective Geometric Communication Structure for Programmable Matter. In *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:3, 2022. doi:10.4230/LIPICS.DISC.2022.47.

26 Sam Kriegman, Douglas Blackiston, Michael Levin, and Josh Bongard. A Scalable Pipeline for Designing Reconfigurable Organisms. *Proceedings of the National Academy of Sciences*, 117(4):1853–1859, 2020. doi:10.1073/PNAS.1910837117.

27 Bruce J. MacLennan. The Morphogenetic Path to Programmable Matter. *Proceedings of the IEEE*, 103(7):1226–1232, 2015. doi:10.1109/JPROC.2015.2425394.

28 Othon Michail, George Skretas, and Paul G. Spirakis. On the Transformation Capability of Feasible Mechanisms for Programmable Matter. *Journal of Computer and System Sciences*, 102:18–39, 2019. doi:10.1016/J.JCSS.2018.12.001.

29 Sanaz Mostaghim, Christoph Steup, and Fabian Witt. Energy Aware Particle Swarm Optimization as Search Mechanism for Aerial Micro-Robots. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–7, 2016. doi:10.1109/SSCI.2016.7850263.

30 Nils Napp, Samuel Burden, and Eric Klavins. Setpoint Regulation for Stochastically Interacting Robots. *Autonomous Robots*, 30(1):57–71, 2011. doi:10.1007/S10514-010-9203-2.

31 Daniel Pickem, Paul Glotfelter, Li Wang, Mark Mote, Aaron Ames, Eric Feron, and Magnus Egerstedt. The Robotarium: A Remotely Accessible Swarm Robotics Research Testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1699–1706, 2017. doi:10.1109/ICRA.2017.7989200.

32 Benoit Piranda and Julien Bourgeois. Designing a Quasi-Spherical Module for a Huge Modular Robot to Create Programmable Matter. *Autonomous Robots*, 42:1619–1633, 2018. doi:10.1007/S10514-018-9710-0.

33 Alexandra Porter and Andréa W. Richa. Collaborative Computation in Self-Organizing Particle Systems. In *Unconventional Computation and Natural Computation*, volume 10867 of *Lecture Notes in Computer Science*, pages 188–203, 2018. doi:10.1007/978-3-319-92435-9_14.

34 Tommaso Toffoli and Norman Margolus. Programmable Matter: Concepts and Realization. *Physica D: Nonlinear Phenomena*, 47(1-2):263–272, 1991. doi:10.1016/0167-2789(91)90296-L.

35 Hongxing Wei, Bin Wang, Yi Wang, Zili Shao, and Keith C.C. Chan. Staying-Alive Path Planning with Energy Optimization for Mobile Robots. *Expert Systems with Applications*, 39(3):3559–3571, 2012. doi:10.1016/J.ESWA.2011.09.046.

36 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 *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, pages 353–354, 2013. doi:10.1145/2422436.2422476.

A Omitted Analysis of Concurrency-Compatibility

This appendix contains the technical material omitted from Section 5 due to space constraints.

Algorithm 2 Expansion-Robust Variant $\mathcal{A}^E$ of Algorithm $\mathcal{A}$ for Amoebot A.

Input: Algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \dots, m\}\}$ satisfying Conventions 1 and 2.
Set $\alpha_0^E : (\exists$ port p of $A : A.\texttt{flag}_p = \text{TRUE}) \to \text{WRITE}(\bot, \texttt{flag}_p, \text{FALSE})$.
for each action $[\alpha_i : g_i \to ops_i] \in \mathcal{A}$ **do**
  Set $g_i^E \leftarrow g_i$ with $N(A)$ replaced by $N^E(A)$ and connections defined w.r.t. $N^E(A)$.
  Set $ops_i^E \leftarrow$ "Do:
    for each port p of A **do** WRITE$(\bot, \texttt{flag}_p, \text{FALSE})$. ▷ Reset own expand flags.
    for each unique neighbor $B \in$ CONNECTED() **do**
      for each port p of B **do** WRITE$(B, \texttt{flag}_p, \text{FALSE})$. ▷ Reset neighbors' expand flags.
    Execute each operation of ops_i with connections defined w.r.t. $N^E(A)$.
    if a PULL or PUSH operation was executed with neighbor B **then**
      for each new port p of A not connected to B **do** WRITE$(\bot, \texttt{flag}_p, \text{TRUE})$.
      for each new port p of B not connected to A **do** WRITE$(B, \texttt{flag}_p, \text{TRUE})$.
    else if an EXPAND operation was successfully executed **then**
      for each new port p of A **do** WRITE$(\bot, \texttt{flag}_p, \text{TRUE})$.
    else if an EXPAND operation failed in its execution **then** undo ops_i."
return $\mathcal{A}^E = \{[\alpha_i^E : g_i^E \to ops_i^E] : i \in \{0, \dots, m\}\}$.

▶ **Lemma 18.** *If amoebot algorithm $\mathcal{A}$ is expansion-corresponding, it is also expansion-robust.*

Proof. To prove termination, suppose to the contrary that all sequential executions of $\mathcal{A}$ starting in C_0 terminate, but there exists some infinite sequential execution $\mathcal{S}^E$ of $\mathcal{A}^E$ starting in C_0^E. Algorithm $\mathcal{A}$ is expansion-corresponding, so there is a sequential execution $\mathcal{S}$ that is identical to $\mathcal{S}^E$, modulo executions of α_0^E. Execution $\mathcal{S}$ terminates by supposition, so $\mathcal{S}^E$ must contain an infinite number of α_0^E executions after its final $\alpha_{i\neq 0}^E$ execution. But α_0^E executions only reset expand flags, and there are only a finite number of amoebots and a constant number of expand flags per amoebot to reset, a contradiction.

Correctness follows from the same observation. Only $\alpha_{i\neq 0}^E$ executions move amoebots and modify variables of $\mathcal{A}$. Since every sequential execution $\mathcal{S}^E$ of $\mathcal{A}^E$ starting in C_0^E represents an identical sequential execution $\mathcal{S}$ of $\mathcal{A}$ starting in C_0 (after removing the α_0^E executions), and since $\mathcal{S}^E$ terminates whenever $\mathcal{S}$ terminates by the above argument, we conclude that they must terminate in configurations that are identical, modulo expand flags. ◀

Before proving that the energy distribution framework preserves expansion-correspondence, we need one helper lemma characterizing established neighbors in $\mathcal{A}^\delta$.

▶ **Lemma 19.** *During an execution of $(\mathcal{A}^\delta)^E$, if an amoebot A has a neighbor $B \in N(A)$ that is IDLE, PRUNING, or a child of A, then $B \in N^E(A)$.*

Proof. Any neighbor $B \in N(A) \setminus N^E(A)$ expanded into $N(A)$ during an EXPAND operation by B, a PUSH operation by B, or a PULL operation by some other amoebot pulling B. Any movement in $(\mathcal{A}^\delta)^E$ occurs in an $(\alpha_i^\delta)^E$ execution, whose guard requires that both

the executing amoebot and all its established neighbors are not IDLE or PRUNING. Thus, regardless of whether B is initiating the movement (an EXPAND or PUSH) or is participating in it (a PULL), B cannot be IDLE or PRUNING when it enters $N(A)$. Any subsequent action execution that could make B IDLE or PRUNING must also reset its expand flags (Algorithm 2, Line 7). So there are never IDLE or PRUNING neighbors in $N(A) \setminus N^E(A)$.

Next consider any child B of A. Amoebot B became a child of A when A adopted it during a $g_{\text{GROWFOREST}}$-supported execution of $\alpha^E_{\text{ENERGYDISTRIBUTION}}$. During this execution, A reset all expand flags of B (Algorithm 2, Line 7). As long as B is a child of A, its expand flags facing A remain reset. Thus, $B \in N^E(A)$. ◀

We can now prove the main lemma of this section.

▶ **Lemma 20.** *For any energy-compatible, expansion-corresponding algorithm $\mathcal{A}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$, the algorithm $\mathcal{A}^\delta$ produced from $\mathcal{A}$ and δ by the energy distribution framework is concurrency-compatible.*

Proof. By Theorem 2, we know that every sequential execution of $\mathcal{A}^\delta$ terminates. It remains to show that $\mathcal{A}^\delta$ satisfies the validity, phase structure, and expansion-robustness conventions.

By supposition, every action $\alpha_i \in \mathcal{A}$ in the original algorithm is valid, i.e., its execution is successful whenever it is enabled and all other amoebots are inactive. Since the guard g_i of α_i is a necessary condition for the energy-constrained version α^δ_i to be enabled, we know this validity carries over to the compute and movement phases of α_i. The only new operations added by the energy distribution framework in the α^δ_i and $\alpha_{\text{ENERGYDISTRIBUTION}}$ actions are CONNECTED operations (which never fail) and READ and WRITE operations involving existing neighbors. All of these must succeed, so every action of $\mathcal{A}^\delta$ is valid.

It is easy to see that $\mathcal{A}^\delta$ satisfies the phase structure convention. Its only movements are in the α^δ_i actions, each of which has at most one movement operation that it executes last. Moreover, the energy distribution framework does not add any LOCK or UNLOCK operations.

It remains to show $\mathcal{A}^\delta$ is expansion-robust, and by Lemma 18, it suffices to show $\mathcal{A}^\delta$ is expansion-corresponding. We first show that if some action of $(\mathcal{A}^\delta)^E$ is enabled for an amoebot A w.r.t. $N^E(A)$, then the corresponding action of $\mathcal{A}^\delta$ is enabled for A w.r.t. $N(A)$. We may safely consider only the guard conditions that depend on an amoebot's neighborhood; all others evaluate identically regardless of neighborhood.

- If $(\alpha^\delta_i)^E$ is enabled for an amoebot A, then A must satisfy g^E_i – i.e., A satisfies the guard g_i of $\alpha_i \in \mathcal{A}$ w.r.t. $N^E(A)$ – and neither A nor its established neighbors can be IDLE or PRUNING. Algorithm $\mathcal{A}$ is expansion-corresponding by supposition, so this implies that A must satisfy g_i w.r.t. $N(A)$ as well. Moreover, Lemma 19 ensures that if there are no IDLE or PRUNING neighbors in $N^E(A)$, there are none in $N(A)$ either.
- Suppose $\alpha^E_{\text{ENERGYDISTRIBUTION}}$ is enabled for an amoebot A because A has an IDLE neighbor or an ASKING child $B \in N^E(A)$, a condition in both $g_{\text{ASKGROWTH}}$ and $g_{\text{GROWFOREST}}$. We know $N^E(A) \subseteq N(A)$, so $\alpha_{\text{ENERGYDISTRIBUTION}}$ must be enabled for A w.r.t. $N(A)$ as well.
- Suppose $\alpha^E_{\text{ENERGYDISTRIBUTION}}$ is enabled for an amoebot A because A has a child $B \in N^E(A)$ whose battery is not full, a condition in $g_{\text{SHAREENERGY}}$. By the same argument as above, we have $N^E(A) \subseteq N(A)$, so $\alpha_{\text{ENERGYDISTRIBUTION}}$ must be enabled for A w.r.t. $N(A)$ as well.

Finally, we show that the executions of any action of $(\mathcal{A}^\delta)^E$ w.r.t. $N^E(A)$ and the corresponding action of $\mathcal{A}^\delta$ w.r.t. $N(A)$ by the same amoebot A are identical. We may safely focus only on the parts of action executions that depend on or interact with an amoebot's neighbors; all others execute identically regardless of neighborhood.

- If A executes an $(\alpha_i^\delta)^E$ action, it emulates the operations of $\alpha_i \in \mathcal{A}$ w.r.t. $N^E(A)$. But algorithm $\mathcal{A}$ is expansion-corresponding by supposition, which immediately implies that an execution of α_i w.r.t. $N(A)$ is identical.
- If A executes an $(\alpha_i^\delta)^E$ action or the GetPruned block of $\alpha^E_{\text{EnergyDistribution}}$, it may update its children's `state` and `parent` variables during Prune( ). By Lemma 19, any child of A in $N(A)$ is also in $N^E(A)$, so the same children are pruned.
- If A executes the GrowForest block of $\alpha^E_{\text{EnergyDistribution}}$, it adopts all its idle neighbors as an active children. Any idle neighbor $B \in N^E(A)$ that A adopts must also be adopted when A executes $\alpha_{\text{EnergyDistribution}}$ since $N^E(A) \subseteq N(A)$. But if there are no idle neighbors in $N^E(A)$ for A to adopt, there cannot be any in $N(A)$ either by Lemma 19. Thus, either the same idle neighbors or no neighbors are adopted.
- If A executes the GrowForest block of $\alpha^E_{\text{EnergyDistribution}}$, it updates any asking children to growing. By Lemma 19, any child of A in $N(A)$ is also in $N^E(A)$, so the same children are updated in $\alpha_{\text{EnergyDistribution}}$.
- If A executes the ShareEnergy block of $\alpha^E_{\text{EnergyDistribution}}$, it transfers an energy unit to one of its children $B \in N^E(A)$ whose battery is not full. We know $N^E(A) \subseteq N(A)$, so B is also a possible recipient of this energy in $\alpha_{\text{EnergyDistribution}}$. ◀

A Fair and Resilient Decentralized Clock Network for Transaction Ordering

Andrei Constantinescu
ETH Zürich, Switzerland

Diana Ghinea
ETH Zürich, Switzerland

Lioba Heimbach
ETH Zürich, Switzerland

Zilin Wang
ETH Zürich, Switzerland

Roger Wattenhofer
ETH Zürich, Switzerland

Abstract

Traditional blockchain design gives miners or validators full control over transaction ordering, i.e., they can freely choose which transactions to include or exclude, as well as in which order. While not an issue initially, the emergence of decentralized finance has introduced new transaction order dependencies allowing parties in control of the ordering to make a profit by front-running others' transactions. In this work, we present the *Decentralized Clock Network*, a new approach for achieving fair transaction ordering. Users submit their transactions to the network's clocks, which run an agreement protocol that provides each transaction with a timestamp of receipt which is then used to define the transactions' order. By separating agreement from ordering, our protocol is efficient and has a simpler design compared to other available solutions. Moreover, our protocol brings to the blockchain world the paradigm of asynchronous fallback, where the algorithm operates with stronger fairness guarantees during periods of synchronous use, switching to an asynchronous mode only during times of increased network delay.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases Median Validity, Blockchain, Fair Ordering, Front-running Prevention, Miner Extractable Value

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.8

Related Version *Full Version*: https://arxiv.org/abs/2305.05206

1 Introduction

The first blockchain, a decentralized distributed digital ledger that records transactions across a network of computers, was introduced in 2008 with Bitcoin by Nakamoto [33]. Blockchains offer a novel way of storing and transferring value in a trustless and secure manner, without the need for intermediaries. Despite their popularity, blockchain adoption was slow, as blockchains were, initially, mainly used to facilitate simple transfers of money between two individuals. However, this changed in 2015 with the introduction of smart contracts on Ethereum [44], allowing for complex digital agreements to be carried out on-chain. Nowadays, smart contracts are the backbone of a rapidly-growing complex ecosystem of decentralized financial applications known as *decentralized finance (DeFi)*. DeFi offers most traditional financial services, including decentralized exchanges, lending protocols, and stablecoins, without relying on financial intermediaries.

The smart contracts that govern DeFi are generally dependent on the transaction order. That is, the outcome of executing a set of transactions depends on their order. As most transactions were simple transfers in the early days, the original blockchain design did not need to pay much attention to transaction ordering. Instead, the power of transaction ordering is concentrated in miners or validators, which can freely choose which transactions to include and how to order them inside each block. Nowadays, block proposers (miners) extract profit from appropriately ordering, including, and excluding transactions during block production. This profit is known as miner (or maximal) extractable value (MEV). MEV accounts for a profit of at least US$ 650M [21] so far. In fact, Flashbots and other transaction relay protocols organized a whole market around ordering transactions.

Front-running Attacks

Most MEV relies on the ability of the attacker to *front-run* the victim's transaction tx. To be specific, the attacker observes a newly generated victim transaction tx in the mempool (the public waiting area for transactions). The attacker then introduces their own transaction tx'. If tx' executes before tx (front-running), the attacker profits at the expense of the victim. So, the attacker may simply bribe the block proposer with a high fee to execute tx' first, even though tx' was only created once tx was already publicly known.

Front-running can be broadly categorized into two types [35]: *tolerant* and *destructive*. Tolerant front-running involves the attacker placing their own transaction before the victim's transaction in the order of execution. This allows the attacker to gain an advantage, such as purchasing an asset at a lower price before the victim can. Such attacks are often seen on decentralized exchanges, where the attacker executes a trade before the victim, reaping the benefits of price changes. Destructive front-running, on the other hand, has the attacker taking out the victim's transaction altogether. Generally, the attacker copies the victim's presumably profitable transaction. If the attacker's transaction executes first, the victim's transaction would no longer execute, at least not as intended.

Our Contribution

We propose the Decentralized Clock Network (DCN), a novel solution for achieving fair transaction ordering. More concretely, our system ensures that, if a transaction tx was sent to the system long enough before transaction tx', then tx' cannot be ordered before tx, i.e., preventing tolerant front-running. In contrast to most previous solutions relying on the blockchain consensus algorithm to determine a relative ordering of the transactions, our approach employs a decentralized network of n nodes, equipped with clocks, resilient to $f < n/3$ byzantine failures to agree on a timestamp for each transaction. These timestamps are subsequently used to determine the order of the transactions inside each block and across blocks. Decoupling timestamping from ordering enables lower latency bounds whilst reducing the complexity of the consensus mechanism.

A blockchain system is synchronous if all messages arrive at the receiver within a known time-bound, and the nodes involved have local clocks that are (almost) perfectly synchronized. However, in times of turmoil, such as when participants are under attack, messages might experience longer delays, or clocks may no longer be aligned with real-time. Such failures are modeled by the asynchronous model. An important novelty in our work is that our protocol is designed to provide guarantees regardless of the network conditions, without knowing in advance which setup to expect. It is designed for the asynchronous model, however, if the network happens to satisfy some synchrony assumptions, which is often the case in

real-world networks, it provides stronger guarantees reflecting in the order obtained. To quantify this effect, we propose a new notion of order fairness, called δ-Median Fairness. Roughly, transactions shall be ordered based on a value that is close to the median of the points in time when honest nodes in the DCN first learn about the transaction. Here, δ is an error parameter, determining the closeness of the estimated median to the true median of the honest timestamps in terms of quantiles. This definition is a stronger version of Honest-Range Fairness (or fair separability, as defined in [46]). When operating under asynchronous conditions, our algorithm achieves f-Median Fairness, which coincides with Honest-Range Fairness in the worst case $n = 3f + 1$, but is stronger otherwise. On the other hand, when the network is synchronous for a sufficient amount of time, our algorithm achieves the superior guarantee of $\lceil f/2 \rceil$-Median Fairness. In both cases, these guarantees are optimal. We add that our protocol sidesteps the attack where relative orders relying on the median can be manipulated by a single byzantine node presented in [27] by ensuring that (1) nodes always agree on some honest timestamp, and (2) with the help of cryptographic primitives, we do not allow nodes, or anyone else, to see the transaction contents before a timestamp is agreed upon.

Related Work

Fair Ordering. Blockchain front-running prevention techniques have been the subject of significant research in recent years. We point the reader to Baum et al. [5] and Heimbach et al. [25] for an overview of these approaches and only discuss the most relevant in the following.

Flashbots [20] and other private relay services, in which transactions are sent directly to a trusted third party for ordering and subsequent forwarding to validators for block inclusion, are widely adopted. While this approach is efficient, it centralizes the transaction ordering process, i.e., introduces a single point of failure, and is often used to front-run as opposed to protect against. In contrast, our approach distributes the transaction ordering responsibility.

In the field of fair transaction ordering, committee-based approaches have been widely studied. Generally, these approaches can be divided into two categories: those that can operate in asynchrony and those that assume partial synchrony, which is a model weaker than synchrony and stronger than asynchrony. To tackle fair ordering in partial synchrony, Pompe is proposed by Zhang et al. [46], Wendy is proposed by Kursawe [28] and Themis is proposed by Kelkar et al. [26]. As opposed to these protocols, the DCN we propose is equipped to handle asynchrony. In particular, Pompe and Themis rely on (partial) synchrony and Wendy assumes the clocks of the nodes are always synchronized.

Kelkar et al. [27] introduce Aequitas, which achieves state-of-the-art fairness properties, but has a significant communication complexity of $\mathcal{O}(n^4)$ in asynchrony. Our agreement protocol achieves in expectation $\mathcal{O}(n^3 \log \Delta)$ message complexity in asynchrony, where Δ denotes the *observed* network delay. We note that this delay does not have to be known a priori, as opposed to classical synchronous protocols.

Quick order fairness, introduced by Cachin et al. [14] achieves $\mathcal{O}(n^3)$ message complexity in asynchrony. While their protocol allows for a node to gain insider information before an ordering is agreed upon, our protocol adds further protection to users as the committee only sees the full transaction after the timestamp is agreed upon. Further, their approach, and the others, only target agreement amongst the permissioned committee, while our design extends to implementing the fair ordering on a permissionless blockchain after agreement has been reached in the permissioned committee.

Agreement Protocols. Achieving agreement on a value subject to some Validity condition, i.e., Byzantine Agreement (BA) [29], is an extensively studied problem in Distributed Computing. In real-world applications, hence also in our setting, it is desirable to expect the Validity condition to carry some meaning, while the classical BA definition only ensures that if honest nodes have the same input value v, they all output v. If this pre-agreement condition is not met, the honest nodes may output an adversarially chosen value. Recent works have focused on achieving more meaningful guarantees, such as ensuring that the honest output is *close* to the honest inputs' median [39], to the k-th lowest honest input [31], or somewhere in the range of honest inputs [42]. These works, however, only focus on the synchronous model. That is, they assume perfectly synchronized clocks and a publicly available upper bound on the network delay. A more realistic setting is the so-called asynchronous model, which drops this assumption, but showcases important limitations: in the asynchronous setting, BA cannot be achieved deterministically [19]. There is still hope, however: randomized asynchronous BA protocols exist [6, 11, 13, 15, 22, 32, 36, 41]; however, without meaningful Validity guarantees if the input space contains more than two values. Another relaxed variant of BA is Approximate Agreement (AA) [3, 18], which offers deterministic protocols that enable honest nodes to output values within the range of their inputs, with the caveat of weakening the Agreement guarantees: honest outputs are ε-close for any predefined $\varepsilon > 0$.

To implement our fair-ordering definition, we propose an asynchronous (randomized) BA protocol with optimal resilience, that achieves Median Validity [31, 40] with optimal-error guarantees, assuming that the inputs are integers. Our lower bound on this error implies that, when the network is asynchronous, and when aiming for optimal resilience, the best one can hope for is obtaining outputs within the range of the honest inputs. We circumvent this problem by designing a protocol whose Validity guarantees scale with the network conditions: if the synchrony assumptions are satisfied for a sufficient amount of time, our protocol will enable honest nodes to agree on a value satisfying the synchronous model's optimal guarantees on Median Validity. Otherwise, our protocol will at least provide Median Validity with optimal guarantees for the asynchronous model, hence the output agreed upon will be within the range of honest values. Designing protocols that achieve simultaneously optimal guarantees in both synchronous and asynchronous networks, has been a topic that attracted increased attention in the recent years in the Distributed Computing literature. There has been a line of works focusing on problems such as Byzantine Agreement [7], Approximate Agreement [23], State Machine Replication [8], and also Multi-Party Computation [4, 9, 17].

2 The Decentralized Clock Network

In this section, we describe the DCN, which consists of a network of nodes equipped with synchronized clocks operating with the objective of providing an accurate and decentralized timestamping service to blockchain transactions. The resulting timestamps are used to determine the ordering of the transactions inside each block, as well as across blocks. The intuition behind using a timestamping service is that, instead of relying on consensus to determine the ordering directly, like in FSS from ChainLink Labs [16], this way the order of the transactions is naturally induced by the timestamps, allowing the complexity of the agreement protocol to be reduced.

High-Level Design

To enable DCN support for ordering transactions on an existing blockchain, the blockchain requires only minor adaptations. In particular, with every submitted transaction, an additional timestamp computed by the DCN is expected. Validators should check for each block

whether timestamps are authentic and whether the ordering induced by the timestamps is respected, rejecting the block otherwise. In order for this check to take place, timestamps computed by the DCN are accompanied by threshold signatures, cryptographic gadgets used to prove that each timestamp was agreed upon by at least one honest node. Nodes in the DCN must not only be trustworthy, but also have good network conditions and be able to handle a large volume of service requests. To ensure the precision and consistency of the nodes' clocks, as well as nodes' high availability, we implement the DCN as a permissioned system, where the identity and public keys of the nodes are known to the validators. Nodes are not intended to change frequently and, by keeping the set of nodes in the system fixed, we can ensure that the nodes are reliable and that the timestamping service is accurate.

Network Model and Assumptions

The DCN consists of n nodes in a fully-connected network, such that any two nodes in the network can communicate through authenticated channels. Nodes can moreover receive external inputs, e.g., transactions from users. Each node comes equipped with a clock. We assume that node clocks are periodically realigned with real-time, which can be achieved through the use of a common external reference, such as UTC time or GPS time.

We consider an adaptive adversary that takes control during the protocol's execution of at most $f < n/3$ nodes, causing them to deviate arbitrarily (even maliciously) from the protocol; i.e., byzantine behavior.

We assume an estimation Δ_{DCN} representing an upper bound on the network delay within the DCN, i.e., messages sent between the nodes *should* be delivered within Δ_{DCN} time. Similarly, we assume an estimation Δ_{EXT} for the upper bound on the external network delay, i.e., for messages sent between users and the nodes in the DCN. We say that the network is synchronous if the message delays are *always* at most Δ_{DCN} and Δ_{EXT}, and the nodes' clocks are perfectly synchronized. If any of these conditions fail at any point, then the network is asynchronous. In our work, we will assume that the network is asynchronous. However, we take into account that an asynchronous assumption is often too pessimistic to model a real-life network. Hence, we aim to offer stronger guarantees during timespans when the network is synchronous, which should be the case most of the time if our estimations Δ_{DCN} and Δ_{EXT} are faithful. We also take into account that real clocks may fail the perfect synchrony assumption; i.e., they may have a small skew S, or their local rate may vary by a factor $\theta = 1 + o(1)$, as described in [30]. However, we assume perfect synchronization for simplicity of presentation, and we will briefly describe how our protocols can be modified to achieve the same synchronous guarantees under the weaker clock synchronization assumptions.

Cryptographic Primitives

As mentioned previously, we employ threshold signatures. In an (ℓ, n)-threshold signature scheme, a public key is known to the n nodes and also to all users and validators. Moreover, each node v knows a unique private key that enables the generation of a partial signature $\sigma_v(m)$ for any message m. The defining property of the scheme is that ℓ partial signatures from distinct nodes for the same message m can be combined into a single signature $\sigma(m)$ that can be verified using the public key. Formally, the scheme should satisfy robustness and non-forgeability (see the full version of [12, Section 2.3.2] for the definitions). For our purposes, we set the threshold $\ell = f + 1$ and choose the BLS scheme [10, 38].

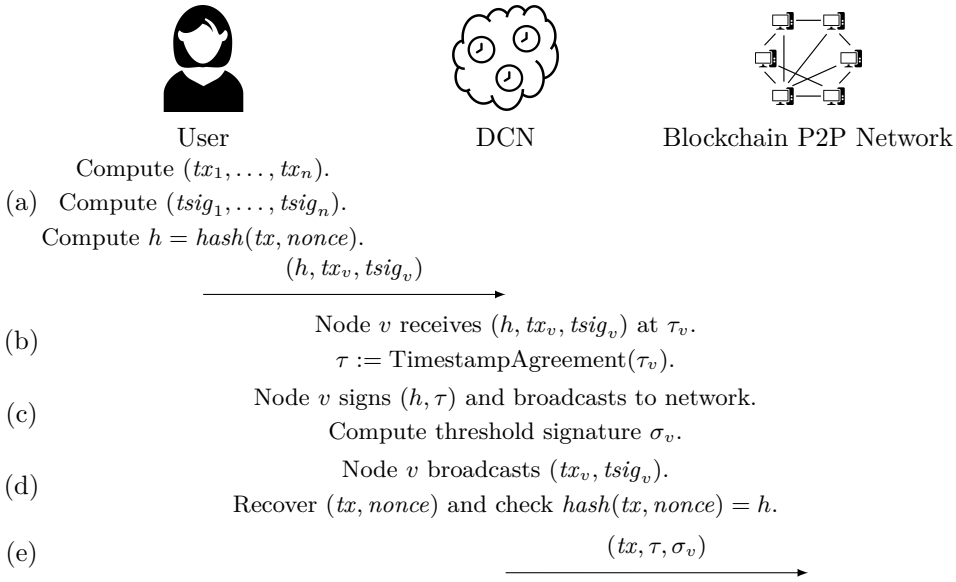


Figure 1 Illustration of the transaction submission (i.e., main) protocol.

Furthermore, we require a secret sharing scheme. In a (k, n)-secret sharing scheme, a secret, such as a user transaction, is divided into n so-called *shares*, one known to each node, such that any k nodes can reconstruct the secret, while any coalition of at most $k-1$ nodes cannot learn anything about it. Formally, the information-theoretic requirement is that any k shares uniquely determine the secret, while any $k-1$ shares must be independent of the secret. Informally, given $k-1$ shares, every possible transaction is equally likely to result in these shares. In our work, we choose $k = f + 1$ and use the Shamir scheme [37].

The Transaction Submission Protocol

In this section, we formally present the protocol used when users submit transactions to the blockchain (cf. Figure 1), which we also refer to as the *main* protocol. Assume a user wants to submit transaction tx, then the following steps are to be followed:

(a) The user generates a random nonce $nonce$. Then, the user splits the pair $(tx, nonce)$ into n shares $(tx_1, \ldots, tx_n)$ using the $(f+1, n)$-secret sharing scheme and signs the shares with their private key to get $(tsig_1, \ldots, tsig_n)$. Subsequently, the user hashes the transaction together with the nonce as $h = hash(tx, nonce)$. Finally, they send to each node v the tuple $(h, tx_v, tsig_v)$.

(b) Each node v receives $(h, tx_v, tsig_v)$ at some time τ_v. Together, the nodes run a *Timestamp Agreement* protocol to agree on a common timestamp τ for transaction tx. The agreement protocol is described in detail in Section 4.

(c) Upon reaching agreement, each node signs (h, τ) and broadcasts the signature to the other nodes. Each node v receives the signatures of (h, τ), verifies them, and uses the at least $f + 1$ valid ones to compute a threshold signature σ_v for the pair (h, τ).

(d) Afterwards, each node v broadcasts their signed share $(tx_v, tsig_v)$ to all other nodes. Each node receives the signed shares, verifies the signatures, and uses the at least $f + 1$ valid shares to recover the pair $(tx, nonce)$. Finally the node checks whether $hash(tx, nonce) = h$, aborting the protocol otherwise.

(e) Each node v now knows tx and submits it timestamped to the blockchain's peer-to-peer (P2P) network as the tuple (tx, τ, σ_v).

The blockchain now operates with tuples of the form (tx, τ, σ) instead of just transactions tx. For each transaction in a block, validators check the threshold signature σ using the public keys of the nodes. Moreover, they also check that transactions are ordered in non-decreasing order by τ inside the block and that the lowest timestamp in the block is no lower than the highest timestamp in the previous block.

We now provide additional intuition for the submission protocol and reasoning behind some of the design considerations. Step (a) describes the user-sided part, while steps (b)–(e) describe the DCN-sided part.

In step (a) the user hashes tx together with a random nonce and sends it to the nodes. The nonce is required to prevent malicious actors from inferring information about tx based on past transaction data; e.g. if a user submits similar transactions periodically, they can be identified by their hash and front-run, e.g., buying ETH every time they receive their paycheck. Moreover, the transaction-nonce pair is split into n shares which are distributed to the n nodes. This allows the DCN to recover the pair $(tx, nonce)$ after agreeing on timestamp τ, check its integrity against the hash h, and submit tx to the blockchain on the user's behalf, preventing users from submitting many timestamping requests without submitting matching transactions to the blockchain, which would be the source of attacks.

In step (b) the DCN agrees on a common timestamp τ for transaction tx using the agreement protocol described later on in Section 4, which is efficient, robust to at most $f < n/3$ byzantine failures in both synchronous and asynchronous settings, and achieves good fairness guarantees, whose definitions we postpone to the next section.

In step (c) the DCN computes threshold signatures for the pair (h, τ) consisting of the transaction hash together with timestamp τ. Any valid such signature can be used to prove that at least $f + 1$ nodes have agreed on it; i.e., at least one honest node.

In step (d) the nodes circulate their shares to recover the pair $(tx, nonce)$. Note that this has to be done after agreeing on the timestamp because otherwise a byzantine node could front-run tx by submitting its own transaction and having agreement happen for it faster than for tx. Moreover, checking the hash of the pair against h is required to prevent dishonest users from sending contradicting shares. Note that steps (c) and (d) can be implemented concurrently, but we chose not to do so for simplicity of exposition.

Finally, in step (e) each node v submits tx together with timestamp τ and threshold signature σ_v to the blockchain, which will handle checking the signatures and ensuring that transactions are ordered by timestamp inside each block and across blocks. Note that different nodes might compute different threshold signatures σ_v, even in the presence of no byzantine nodes, because of the choice of which individual signatures to include in σ_v, but any valid such signature is enough to certify the tuple (tx, τ, σ_v). We further note that validators will of course check that the transaction tx is only executed once.

We state our protocol's guarantees in the theorem below, which we prove in Section 5. We provide a formal definition for the term *fair timestamp* in Section 3.

▶ **Theorem 1.** *The transaction submission protocol achieves the following properties:*

- *(Honest-User Liveness) If a transaction is sent by an honest user, it gets processed and submitted to the mempool eventually. Moreover, if the honest user's messages reach the nodes within Δ_{EXT} time and the synchrony assumptions hold inside the DCN for an additional Δ_{DCN} time, the transaction gets submitted within expected $\mathcal{O}(\log \Delta_{EXT})$ communication rounds.*
- *(Integrity) If a transaction gets submitted to the mempool, the process was initiated by some user.*

- *(Unique Timestamp) If a transaction gets submitted to the mempool by two nodes, with timestamps τ and τ', then $\tau = \tau'$.*
- *(Fair Timestamp) If a transaction gets submitted to the mempool with timestamp τ, then τ is a* fair *timestamp.*

3 Timestamp Agreement and The Fairness Guarantees

Nodes in the DCN need to agree on a timestamp for each transaction. This problem reduces to achieving asynchronous Byzantine Agreement (aBA), with a special Validity condition, which will allow us to argue why transactions are ordered in a fair manner. We recall the classical definition of aBA, which requires the following properties: *(Weak Validity)* If all honest nodes have input τ, no honest node outputs $\tau' \neq \tau$; *(Agreement)* If two honest nodes output τ and τ', then $\tau = \tau'$; *(Termination)* Every honest node outputs with probability 1.

While aBA is an essential building block in distributed computing, it comes with many limitations. We first note the seminal result of [19], which proves that fault-tolerant aBA, even with binary inputs, cannot be solved deterministically. There is still hope however, as the distributed computing literature offers plenty of randomized aBA protocols [6, 11, 13, 15, 22, 32, 36, 41].

Unfortunately, there is another limitation that prevents us from directly applying existing aBA protocols to our setting, standing in its Weak Validity condition: this only ensures that honest nodes agree on an honest input if they joined aBA with the same input. This pre-agreement condition is a very strong requirement in our setting, and hence nodes may often end up agreeing on timestamps proposed by corrupted nodes. Such timestamps may be too low or too high, preventing us from ensuring any kind of fair ordering. We add that achieving a stronger condition that requires the honest nodes to always agree on some honest node's input is impossible, as one cannot distinguish between an honest node and a byzantine node following the protocol correctly, but with a corrupted input.

Meaningful Timestamps. Fortunately, there are still a few Validity variations we can consider. In the following definitions, we will make use of the timestamps that the nodes record when receiving messages from the user. We need to consider that, if the user is dishonest, some honest nodes might not hold such a timestamp. Note that there is at least one honest node who has received a message from this user (otherwise the user is essentially not sending a transaction). Then, let $\tau_{\max}$ denote the latest point in time recorded by an honest node when receiving this user's message. In the definitions, we assume that, if an honest node does not receive such a message, its input is $\tau_{\max}$. We stress that this assumption is strictly for simplicity of presentation and is not used in our protocols or their analysis.

With this convention in mind, we may provide stronger Validity definitions. In our setting, ensuring that honest nodes' outputs are in their inputs' range is already meaningful (*Honest-Range Validity*). This enables the order fairness definition below, discussed in [46].

▶ **Definition 2** (Honest-Range Fairness). *Let tx and tx' denote two transactions. If all honest nodes receive the hash of tx before any honest node receives the hash of tx', then tx will be ordered before tx'.*

Honest-Range Validity has been studied in the synchronous setting [42]. In the asynchronous setting, however, this condition has only been considered under much weaker Agreement requirements, which allow the honest outputs to be ε-close for some predefined $\varepsilon > 0$; see [3].

One could hope that a stronger order-fairness definition is possible. Our first attempt is as follows: if, at some time τ, most honest nodes have received the hash of some transaction tx, while most honest nodes are yet to receive the hash of some transaction tx', then tx should be ordered before tx'. We express this condition with the help of the medians of the honest nodes' receipt timestamps:

▶ **Definition 3** (Median Fairness). *Suppose the hashes of transactions tx and tx' are received by the honest nodes at times $\tau_1 \leq \tau_2 \leq \ldots \leq \tau_{n-f}$ and resp. $\tau'_1 \leq \tau'_2 \leq \ldots \leq \tau'_{n-f}$. Then, if $\tau_\mu < \tau'_\mu$, where $\mu = \lceil (n-f)/2 \rceil$ denotes the index of the median, tx will be ordered before tx'.*

To achieve this order fairness definition, we need honest nodes to agree on the median of their timestamps. Consider the (δ-Median Validity) condition below, introduced by Stolz and Wattenhofer in [39], for $n > 3f$.

- (δ-Median Validity) Assume the honest inputs are arranged in non-decreasing order in an array T, and T_i is the i-th value in T. If an honest node outputs τ, then $\tau \in [T_{\mu-\delta}, T_{\mu+\delta}]$ (i.e., τ is δ-positions-close to T_μ), where $\mu = \lceil (n-f)/2 \rceil$.

Then, Median Fairness requires 0-Median Validity. This definition however cannot be achieved even in a synchronous network, as stated in Lemma 4, following directly from [31,39].

▶ **Lemma 4.** *If $n > 3f$ and $\delta < \lceil f/2 \rceil$, there is no synchronous protocol achieving Termination and δ-Median Validity.*

We therefore weaken our Median Fairness definition to allow some error.

▶ **Definition 5** (δ-Median Fairness). *Suppose the hashes of transactions tx and tx' are received by the honest nodes at times $\tau_1 \leq \tau_2 \leq \ldots \leq \tau_{n-f}$ and $\tau'_1 \leq \tau'_2 \leq \ldots \leq \tau'_{n-f}$ respectively. Let $\mu = \lceil (n-f)/2 \rceil$ denote the index of the median. Then, if $\tau_{\mu+\delta} < \tau'_{\mu-\delta}$, transaction tx will be ordered before transaction tx'.*

We note that δ-Median Validity has only been considered in the synchronous model [31,39], meaning that even the slightest increased network delay may cause the protocols of [31,39], which achieve δ-Median Validity, to completely fall apart. This motivates us to study δ-Median Validity in the asynchronous model. First, we show a lower bound on the δ achievable for the asynchronous case. Later on, we will further show this bound to be tight.

▶ **Lemma 6.** *If $n > 3f$ and $\delta < f$, there is no asynchronous protocol achieving Termination and δ-Median Validity.*

Proof. We assume that there is a protocol Π achieving δ-Median Validity and Termination. Let $\mu = \lceil (n-f)/2 \rceil$, and let v denote an honest node. The input value of node v will be $2f+1$. We define the following scenarios:

(a) The $n-f$ honest nodes have inputs $f+1, f+2, \ldots n$, and the corrupted parties do not participate in the protocol. Then, v must output a value in $[f+\mu-\delta, f+\mu+\delta]$.

(b) The $n-f$ honest nodes have inputs $1, 2, \ldots, n-f \geq 2f+1$. The f corrupted nodes follow the protocol correctly with inputs $n-f+1, n-f+2, \ldots, n$, while the messages of the honest nodes holding the f lowest inputs are delayed. Here, v should output a value in $[\mu-\delta, \mu+\delta]$. However, since from node v's perspective, this scenario is indistinguishable from Scenario (a), v must output a value in $[\mu-\delta, \mu+\delta] \cap [f+\mu-\delta, f+\mu+\delta] = [f+\mu-\delta, \mu+\delta]$.

(c) The $n-f$ honest nodes have inputs $2f+1, 2f+2, \ldots, n+f$. The f corrupted nodes follow the protocol correctly with inputs $f+1, f+2, \ldots, f$, while the messages of the f honest nodes holding the f highest inputs are delayed. Here, v should output a value in

$[2f + \mu - \delta, 2f + \mu + \delta]$. Note that, for node v, this scenario is indistinguishable from Scenario (a) and Scenario (b). Therefore, node v must output a value in $[f + \mu - \delta, \mu + \delta] \cap [2f + \mu - \delta, 2f + \mu + \delta] = [2f + \mu - \delta, \mu + \delta]$.

Since $\delta < f$, we obtain that $\mu+\delta < \mu+f < 2f+\mu-\delta$, therefore the interval $[2f+\mu-\delta, \mu+\delta]$ containing node v's output is empty. This contradicts that Π achieves Termination. ◀

Lemma 6 showcases an important limitation, namely, in a purely asynchronous network, if $n = 3f + 1$, one can only hope to achieve Honest-Range Validity, as in this case f-Median Validity degenerates to Honest-Range Validity. We note here that previous work has shown that a single byzantine node can manipulate the median [27]. However, as timestamps satisfying f-Median Validity are still in the honest range and as transactions are not visible during ordering, we do not see it as a threat.

Defining Timestamp Agreement. To mitigate the limitation posed by Lemma 6, we take into account that real-world networks are not as unreliable as the asynchronous model. Hence, we aim to provide better guarantees if the network *happens to be synchronous*. We investigate whether we can achieve best-of-both-worlds guarantees, in line with many recent works [4, 7, 9, 17, 23]. That is, we investigate whether there is an asynchronous protocol ensuring f-Median Validity that can additionally offer the stronger guarantee of $\lceil f/2 \rceil$-Median Validity if the network *happens to be synchronous* for sufficient time. Therefore, we introduce the following variant of aBA.

▶ **Definition 7** (Timestamp Agreement)**.** *An n-nodes protocol, where each node may hold an integer timestamp as input, achieves Timestamp Agreement (TA) if, even when f of the nodes are corrupted, it achieves Agreement, f-Median Validity, and the following hold:*

- *if all honest nodes hold inputs, then all honest node obtain outputs with probability* 1;
- *if less than $f + 1$ honest nodes hold inputs, then no honest node obtains output;*
- *if the synchrony assumptions hold for a sufficient amount of time and all honest parties receive their inputs accordingly, then $\lceil f/2 \rceil$-Median Validity is achieved.*

We note that we have proposed this definition taking into account that the user is not necessarily honest, and hence may not provide all honest nodes with inputs. If this is the case, our protocol still maintains f-Median Validity and Agreement. For the timestamp submission protocol, this implies that, if a dishonest user's transaction gets submitted to the chain, then the unique timestamp assigned to it still fits our f-Median Fairness definition. Hence, such adversarial behavior does not bring the dishonest user any real advantage.

We may now also define the term *fair* timestamp, used in Theorem 1: it is a timestamp satisfying f-Median Validity, and, if synchrony assumptions hold, $\lceil f/2 \rceil$-Median Validity.

4 The Timestamp Agreement Protocol

In this section, we present our protocol achieving Timestamp Agreement secure against $f < n/3$ byzantine corruptions. Formally, we obtain the result below. Recall once again that there is no deterministic protocol achieving asynchronous Timestamp Agreement, a fact following directly from FLP [19].

▶ **Theorem 8.** *There is an n-nodes randomized protocol Π_{TA} achieving TA resilient against $f < n/3$ byzantine corruptions. Π_{TA} has expected round complexity $\mathcal{O}(\log(\tau_{\max} - \tau_{\min}))$, where $\tau_{\min}$ and $\tau_{\max}$ denote the lowest and the highest honest inputs respectively. To achieve $\lceil f/2 \rceil$-Median Validity, the synchrony assumptions must to hold for $\Delta_{EXT} + \Delta_{DCN}$ time.*

Our protocol Π_{TA} consists of three steps. First, each node obtains a value satisfying δ-Median Validity. This is the only step where synchrony assumptions are required to achieve $\delta = \lceil f/2 \rceil$ instead of $\delta = f$. In the second step, nodes obtain very close values (they agree up to an error of $\varepsilon < 0.5$) within the range of values that honest nodes obtained in the first step. Agreement is then achieved in the last step, where each node decides whether to round its value obtained in the second step up or down. This will be done using aBA on the rounding option's parity. In the following, we describe each step of Π_{TA} in detail.

Step 1: δ-Median Validity. We first design a protocol Π_{init} that only focuses on achieving δ-Median Validity (while Agreement is covered by the subsequent steps).

Concretely, nodes send their input value to every party. To obtain a good estimation on the honest inputs' median, the nodes aim to receive as many honest inputs as possible. If the network is asynchronous, one may only expect to receive $n - f$ values. On the other hand, if the network is synchronous, and the user initiated the transaction at some time τ, all honest inputs are received by time $\tau + \Delta_{\mathsf{EXT}} + \Delta_{\mathsf{DCN}}$. Then, nodes wait until they have received timestamps from at least $n - f$ nodes, and, until at least $\Delta_{\mathsf{EXT}} + \Delta_{\mathsf{DCN}}$ time has passed since they have received the user's message. This way, if the synchrony assumptions hold, every honest timestamp is received.

Hence, each node v collects $n - f + k$ timestamps, where $0 \leq k \leq f$, and arranges them in an array R in non-decreasing order. If the network is synchronous, at most k of these values are corrupted. These may be lower than any honest input, hence shifting the honest median with at most k positions to the right, or higher than any honest input. Therefore, the honest median is in the subarray $R_\mu, R_{\mu+1}, \ldots, R_{\mu+k}$, where R_i denotes the i-th lowest value in R and $\mu = \lceil (n-f)/2 \rceil$. Then, to obtain a value that is $\lceil f/2 \rceil$-positions-close to the median, v outputs $\tau_\mu := R_{\mu+\lfloor k/2 \rfloor}$, i.e., the median of the subarray $R_\mu, R_{\mu+1}, \ldots, R_{\mu+k}$.

If the synchrony assumptions fail, however, the $n - f + k$ values from R might come from f corrupted nodes, and $n - 2f + k$ honest nodes. The $f - k$ missing honest timestamps provide the corrupted nodes with more power: shifting the honest median f positions to the right, or $f - k$ positions to the left. Regardless, the chosen output $\tau_\mu := R_{\mu+\lfloor k/2 \rfloor}$ still ensures that f-Median Validity holds.

We formally present the code of Π_{init} below. We note that this is the only step requiring synchrony for achieving $\lceil f/2 \rceil$-Median Validity. In order to achieve the same guarantee even if the nodes' clocks are not perfectly synchronized, we may replace the waiting time by $\theta \cdot (\Delta_{\mathsf{EXT}} + \Delta_{\mathsf{DCN}})$, to ensure that the fastest node waits long enough.

Protocol Π_{init}

Code for node v with input timestamp τ_{in}

1: Send your input τ_{in} to all nodes.
2: After at least $\Delta_{\mathsf{EXT}} + \Delta_{\mathsf{DCN}}$ time, and when $n - f + k$ timestamps ($0 \leq k \leq f$) are received:
3: $R :=$ an array containing the timestamps received, ordered non-decreasingly.
4: Output $\tau_\mu := R_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor}$.

We may now state and prove the properties of Π_{init}.

The next property enables us to ensure safety guarantees even when the user initiating the process is dishonest. It follows immediately from line 2 of Π_{init}.

▶ **Lemma 9.** *If less than $f + 1$ honest nodes provide inputs τ_{in}, then no honest node outputs. Otherwise, if each honest node provides an input τ_{in}, then all honest nodes output τ_μ.*

The following lemmas show that the nodes indeed obtain values satisfying the desired Validity guarantees.

▶ **Lemma 10.** *If an honest node outputs a timestamp τ_μ, then τ_μ satisfies f-Median Validity.*

Proof. If an honest node v has obtained a timestamp τ_μ, then it has received $n - f + k$ values, where $0 \leq k \leq f$. Out of these, at least $n - 2f + k$ values are honest.

Let T denote the array of honest inputs arranged in non-decreasing order. We show that $T_{\lceil (n-f)/2 \rceil - f} \leq \tau_\mu = R_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor} \leq T_{\lceil (n-f)/2 \rceil + f}$.

For the upper bound, note that R may miss $f - k$ out of the values in T, hence at most $f - k$ of the values T_i with $i \leq \lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor$. This implies that $R_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor} \leq T_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor + (f-k)} \leq T_{\lceil (n-f)/2 \rceil + f}$.

For the lower bound, note that R contains at most f corrupted values, hence at most f additional values that are lower than $T_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor}$. Then, we obtain that $R_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor} \geq T_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor - f} \geq T_{\lceil (n-f)/2 \rceil - f}$. ◀

We now show that Π_{init} achieves $\lceil f/2 \rceil$-Median Validity if the synchrony assumptions hold, using a similar argument to the proof of Lemma 11. The key difference is that at least $n - f$ of the values received are honest (as opposed to $n - 2f + k$).

▶ **Lemma 11.** *If all honest nodes obtain inputs τ_{in} and join Π_{init} between time τ_{start} and time $\tau_{start} + \Delta_{EXT}$, and the synchrony assumptions hold until time $\tau_{start} + \Delta_{EXT} + \Delta_{DCN}$, then all honest nodes output timestamps τ_μ satisfying $\lceil f/2 \rceil$-Median Validity.*

Proof. We first show that all honest timestamps are received by time $\tau_{\mathsf{start}} + \Delta_{\mathsf{EXT}} + \Delta_{\mathsf{DCN}}$. Each honest node sends its input to all other nodes by time $\tau_{\mathsf{start}} + \Delta_{\mathsf{EXT}}$. Since the network is synchronous, these values are received within Δ time, hence by time $\tau_{\mathsf{start}} + \Delta_{\mathsf{EXT}} + \Delta_{\mathsf{DCN}}$. Then, since all honest nodes start the execution of the protocol at time at least τ_{start}, the protocol ensures that each honest node waits until time at least $\tau_{\mathsf{start}} + \Delta_{\mathsf{EXT}} + \Delta_{\mathsf{DCN}}$, and hence receives all honest timestamps.

Then, for every honest node, R contains all honest values, and $0 \leq k \leq f$ values from corrupted nodes. If T denotes the array of honest timestamps arranged in non-decreasing order, we need to show that $T_{\lceil (n-f)/2 \rceil - \lceil f/2 \rceil} \leq R_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor} \leq T_{\lceil (n-f)/2 \rceil + \lfloor f/2 \rfloor}$.

We first focus on the upper bound: since R contains all values T_i, and $k \leq f$, the inequality $R_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor} \leq T_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor} \leq T_{\lceil (n-f)/2 \rceil + \lfloor f/2 \rfloor}$ holds.

For the lower bound, we note that R contains at most $k + \mu + \lfloor k/2 \rfloor$ values lower than $T_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor}$. Out of these $k + \mu + \lfloor k/2 \rfloor$ values, at most k are corrupted. This means that $R_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor} \geq T_{\lceil (n-f)/2 \rceil + \lfloor k/2 \rfloor - k} = T_{\lceil (n-f)/2 \rceil - \lceil k/2 \rceil} \geq T_{\lceil (n-f)/2 \rceil - \lceil f/2 \rceil}$, which concludes our proof. ◀

Step 2: Agreement up to a small error. Honest nodes have obtained timestamps τ_μ satisfying δ-Median Validity via Π_{init}. We now take a step towards achieving Agreement. We make use of an asynchronous protocol Π_{AA} achieving Approximate Agreement [3]. That is, Π_{AA} ensures that, for any given $\varepsilon > 0$, honest nodes obtain ε-close values τ_{AA} within the range of their values τ_μ (maintaining δ-Median Validity). Lemma 12 states the properties of Π_{AA}, and follows directly from [3]. In our case, any constant $\varepsilon < 0.5$ suffices.

▶ **Lemma 12.** *If an honest node outputs τ_{AA}, then τ_{AA} is within the range of timestamps τ_μ obtained by honest nodes in Π_{init}. If two honest nodes output τ_{AA} and τ'_{AA}, then $|\tau_{AA} - \tau'_{AA}| < \varepsilon < 0.5$. In addition, if less than $f + 1$ honest nodes hold timestamps τ_μ, then no honest node outputs; while if all honest nodes hold timestamps τ_μ, then all honest nodes output.*

The guarantee on obtaining outputs only when at least $f+1$ honest nodes participate follows from the fact that Π_{AA} requires nodes to wait for messages from $n-f$ distinct nodes. In addition, properties on honest nodes' outputs (if any) are ensured even if not all honest nodes participate since Π_{AA} is an asynchronous protocol. Concretely, this is because this setting is indistinguishable from a scenario where the non-participating honest nodes are simply delayed.

We add that Π_{AA} does not make any assumption on the range of honest values τ_μ to achieve these guarantees. It runs in iterations allowing the honest values to converge. If all honest nodes hold inputs τ_μ and range size of these inputs is Δ_μ ($\leq$ the difference between the times when the transaction hash is delivered to the honest nodes), then Π_{AA} runs for $\mathcal{O}(\log(\Delta_\mu/\varepsilon))$ iterations. Each iteration consists of a constant number of communication rounds, and incurs message complexity $\mathcal{O}(n^3)$. Therefore, the round complexity of Π_{AA} is $\mathcal{O}(\log \Delta_\mu)$, and the message complexity is $\mathcal{O}(n^3 \log \Delta_\mu)$.

Step 3: Rounding. Honest nodes have obtained ε-close values τ_{AA} satisfying δ-Median Validity. As depicted in Figure 2, since $\varepsilon < 0.5$, the range of honest values τ_{AA} either:
(a) contains an even integer α such that $|\alpha - \tau_{\mathsf{AA}}| < 0.5$ for all honest values τ_{AA};
(b) contains an odd integer α such that $|\alpha - \tau_{\mathsf{AA}}| < 0.5$ for all honest values τ_{AA};
(c) is between two integers: $\alpha \leq \tau_{\mathsf{AA}} \leq \alpha + 1$ for all honest values τ_{AA}.

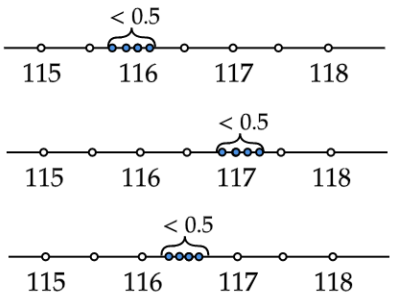


Figure 2 A few examples of possible outputs obtained in Π_{AA}. In the top and middle examples, representing cases (a) and (b) respectively, honest outputs are close to a single integer. In the bottom example, representing case (c), some honest outputs are closer to 116, while some are closer to 117.

Then, the problem of achieving Agreement comes down to enabling the honest nodes to choose between rounding down or rounding up their values τ_{AA}. Making this decision for cases (a) and (b) is trivial: honest nodes simply round their value τ_{AA} to the closest integer. Case (c), however, requires solving aBA. We therefore employ the randomized protocol Π_{aBA} of [32] that achieves aBA with binary inputs in expected round complexity $\mathcal{O}(1)$, with message complexity $\mathcal{O}(n^2)$. Note that we do not use aBA to decide on rounding either up or down, since this would break Agreement in cases (a) and (b). Instead, we use aBA to decide on the parity of the final rounding option. Once again, we note that if the user is dishonest and not all honest nodes were able to reach this stage, protocol Π_{aBA} still offers guarantees. Namely, if less than $f+1$ honest nodes have reached this stage, then no honest node obtains an output (as honest nodes are forced to wait for messages from $n-f$ distinct nodes). Otherwise, if honest nodes obtain outputs, these outputs still satisfy Weak Validity and Termination. This is the case even if not all honest nodes participate, since such a setting is indistinguishable from a scenario where the non-participating honest nodes' messages are simply delayed, and the guarantees of Π_{aBA} hold under asynchrony.

Each node v that has obtained a timestamp τ_{AA} picks two integers α and $\alpha+1$ such that $\alpha \leq \tau_{\mathsf{AA}} < \alpha+1$. Out of these two values, v picks the one that is closer to its τ_{AA} as an initial rounding option, denoted by β. Then, v joins Π_{aBA} with input b, representing the parity of β, and may obtain output b'. If $b'=b$, it outputs β, and otherwise it outputs its second rounding option. In cases (a) and (b), honest nodes pick the same value β. They join Π_{aBA} with the same input bit b, and Weak Validity ensures they output $b'=b$. Therefore, if sufficiently many honest nodes reached this stage, all participating honest nodes output β, which still satisfies δ-Median Validity. In case (c), all honest nodes that reach this stage pick the same value α. In this case, because the input timestamps τ_{in} are integers, both α and $\alpha+1$ satisfy δ-Median Validity. Even if honest nodes make a different choice for β, Π_{aBA} allows them to decide on the same bit b', hence they output the same rounding option.

We may now provide the formal code of our Timestamp Agreement protocol Π_{TA}. We define the constant $\varepsilon = 0.49$, but any choice of $\varepsilon < 0.5$ suffices.

Protocol Π_{TA}

Code for node v receiving a transaction at time τ_{in}

Join Π_{init} with input τ_{in}. Upon obtaining τ_μ via Π_{init}:
  Join $\Pi^{\varepsilon}_{\mathsf{AA}}$ with input τ_μ. Upon obtaining output τ_{AA} in $\Pi^{\varepsilon}_{\mathsf{AA}}$:
    Let α be an integer such that $\alpha \leq \tau_{\mathsf{AA}} < \alpha+1$.
    If $\tau_{\mathsf{AA}} - \alpha < \alpha + 1 - \tau_{\mathsf{AA}}$, set $\beta = \alpha$ and $\beta' = \alpha+1$.
    Otherwise, set $\beta = \alpha+1$ and $\beta' = \alpha$.
    Set $b=0$ if β is even, and $b=1$ if β is odd.
    Join Π_{aBA} with input b. Upon obtaining output b' via Π_{aBA}:
      If $b = b'$, set $\tau_{\mathsf{out}} = \beta$. Otherwise, set $\tau_{\mathsf{out}} = \beta'$. Output τ_{out} and terminate.

We now focus on proving Theorem 8. In Lemma 13, we show that Π_{TA} indeed achieves Timestamp Agreement. Then, Lemma 14 focuses on the round complexity. The requirement of synchrony assumptions holding only for $\Delta_{\mathsf{EXT}} + \Delta_{\mathsf{DCN}}$ in order to achieve $\lceil f/2 \rceil$-Median Validity is given by Π_{init}, since the subsequent steps of Π_{TA} are fully asynchronous.

▶ **Lemma 13.** *If less than $f+1$ honest nodes hold inputs τ_{in}, then no honest node outputs.*

Otherwise, honest nodes that output have obtained the same value τ_{out} satisfying δ-Median Validity, with $\delta = \lceil f/2 \rceil$ if the synchrony assumptions held for $\Delta_{EXT} + \Delta_{DCN}$ time at the beginning of the protocol's execution, and $\delta = f$ otherwise.

In addition, if all honest nodes hold inputs τ_{in}, then all honest nodes output.

Proof. Lemma 9 ensures that $f+1$ honest nodes holding inputs τ_{in} are necessary in order to obtain outputs. In the following, we assume this was the case.

Lemma 12 ensures that honest nodes obtaining outputs (meaning all honest nodes if all of them had inputs τ_{in}) have obtained ε-close approximations τ_{AA} for $\varepsilon < 0.5$. These approximations are within the range of honest values τ_{in}, and hence satisfy δ-Median Validity, as ensured by Lemma 10 and Lemma 11.

Then, we need to consider two cases: when all obtained honest approximations are between two consecutive integers, and when some honest approximations are lower than an integer, while some are higher.

If there is some integer γ such that $\gamma \leq \tau_{\mathsf{AA}} < \gamma+1$ for all obtained honest approximations τ_{AA}, then honest nodes obtain $\alpha = \gamma$ and $\alpha + 1 = \gamma + 1$. Regardless of the chosen β and bit b, honest nodes obtain in Π_{aBA} the same bit b' which refers to the same value: either γ for all honest nodes that reached this stage, or $\gamma+1$ for all honest nodes that reached this stage.

Hence, these honest nodes output the same timestamp. It remains to show that the output timestamp is in the range of honest inputs. If all honest nodes that reached this stage have obtained $\tau_{\mathsf{AA}} = \gamma$, then they joined Π_{aBA} with the same input b representing γ's parity, and hence they output the same γ in the honest range according to Lemma 12. Otherwise, if at least one honest node has obtained $\gamma < \tau_{\mathsf{AA}} < \gamma + 1$, we take into account that the honest inputs are integers. Lemma 12 then implies that both γ and $\gamma + 1$ are in the honest inputs' range.

Otherwise, there is some integer γ such that $\gamma \leq \tau_{\mathsf{AA}} < \gamma+1$ for some honest approximation τ_{AA} and $\gamma + 1 \leq \tau'_{\mathsf{AA}} < \gamma + 2$ for some honest approximation τ_{AA}. Note that, in this case, Lemma 12 ensures that $\gamma + 1$ is in the range of the honest nodes' inputs. In addition, since Lemma 12 ensures $\tau'_{\mathsf{AA}} - \tau_{\mathsf{AA}} < 0.5$, both $\gamma + 1 - \tau_{\mathsf{AA}} < 0.5$ and $\tau'_{\mathsf{AA}} - (\gamma + 1) < 0.5$ hold. This applies to all honest nodes that reached this stage: namely, all these honest nodes choose the same $\beta = \gamma + 1$ and therefore join Π_{aBA} with the same bit b. Then, Π_{aBA} ensures all honest nodes that reached this stage output $b' = b$ and output $\gamma + 1$.

If all honest nodes had inputs τ_{in}, all honest nodes have obtained outputs in Π_{aBA}, and therefore all honest nodes output in Π_{TA}. ◀

The round complexity of Π_{TA} follows from the fact that Π_{AA} ensures termination within $\mathcal{O}(\log(\tau_{\max} - \tau_{\min}))$ rounds, if honest nodes' inputs are between $\tau_{\min}$ and $\tau_{\max}$, while Π_{aBA} ensures termination within expected constant time.

▶ **Lemma 14.** *If all honest nodes hold inputs τ_{in}, then honest nodes output within expected $\mathcal{O}(\log(\tau_{\max} - \tau_{\min}))$ rounds, where $\tau_{\min}$ and $\tau_{\max}$ denote the lowest and the highest honest inputs respectively (hence $\mathcal{O}(\log \Delta_{EXT})$ rounds if the synchrony assumptions are satisfied).*

5 Analysis of the Main Protocol

We now formally prove the properties of the transaction submission protocol. In particular, we prove Theorem 1.

▶ **Lemma 15** (Honest-User Liveness). *If a transaction tx is sent by an honest user, it gets processed and submitted to the mempool eventually, and, if the user's messages reach the nodes within Δ_{EXT} time and the synchrony assumptions hold inside the DCN for an additional Δ_{DCN} time, the transaction get submitted within expected $\mathcal{O}(\log \Delta_{EXT})$ communication rounds.*

Proof. Since tx was sent by an honest user, all honest nodes receive the necessary messages to join Π_{TA}, and hence they obtain a timestamp τ. Then all honest nodes obtain τ and send their shares to the other nodes. Since $f + 1$ shares are necessary to reconstruct tx and the shares are signed by the user (therefore the corrupted nodes cannot send corrupted shares), the honest nodes are able to reconstruct the transaction and submit it to the mempool. The round complexity follows from Lemma 14, and from the fact that the main protocol only adds a constant number of communication rounds over Π_{TA}. ◀

▶ **Lemma 16** (Integrity). *If a transaction tx gets submitted to the mempool, the process was initiated by some user.*

Proof. Submitting a transaction to the mempool requires signatures from $f + 1$ nodes, hence from at least one honest node. This honest node only signs if it has obtained output in the invocation of Π_{TA} corresponding to the hash of tx. This means that honest nodes have joined this execution of Π_{TA}, hence they have received input from some user. ◀

▶ **Lemma 17** (Unique Timestamp). *If a transaction tx gets submitted to the mempool with timestamps τ and τ', then $\tau = \tau'$.*

Proof. Assume that $\tau \neq \tau'$. First, note that timestamps are obtained via Π_{TA}, which assigns tx a unique timestamp by the Agreement property. Therefore, during an invocation of the main protocol for tx, all honest nodes obtain the same timestamp τ. Since $f + 1$ signatures are required for the transaction to be submitted along with its timestamp, the corrupted parties are unable to submit tx to the mempool on their own. Hence, if $\tau \neq \tau'$, there must be an honest party that has signed τ', which happened through a different invocation of the main protocol, hence for a different transaction (ensured by the transaction's nonce). ◀

▶ **Lemma 18** (Fair Timestamp). *If a transaction tx gets submitted to the mempool with timestamp τ, then τ is a* fair *timestamp.*

Proof. Since tx was assigned a timestamp obtained via Π_{TA}, all honest parties have assigned a fair timestamp τ to tx, i.e., satisfying f-Median Validity or $\lceil f/2 \rceil$-Median Validity, depending on the network conditions and on the user's honesty. Then, since $f+1$ signatures are required for tx to be submitted, its timestamp was signed by an honest party, hence it is fair. ◀

6 Discussion

Front-running Resistance. The DCN effectively prevents tolerant front-running, i.e., the attacker's transaction executing before the victim's transaction. Transactions are ordered according to the timestamps returned by the DCN, which fulfill δ-Median Validity. Still, transactions submitted close to each other in time could receive the same timestamp, in which case the validator picks an order, or receives timestamps in the opposite order of the actual submission times. However, this is not an issue, as the transaction contents are hidden until the timestamp is agreed upon by the nodes. Thus, tolerant front-running, which, to be effective, requires the attacker to know the contents of the victim's transaction, is prevented.

The DCN does not address destructive front-running. To be more easily integrateable in the current blockchain infrastructure, the permissioned DCN only supplies the timestamp but does not interfere with the blockchain's consensus. Destructive front-running, thus, remains possible, as the block proposer (miner) could choose not to include the transaction.

Censorship Resistance. We note that by using timestamps as a decision factor when including transactions in a block, transactions become in some sense *block-bound*. Thus, a transaction can become temporarily censored if the block proposer does not include the transaction and the transaction's timestamp is too low to be included in future blocks. Further, under an asynchronous network, transactions may get lost solely due to messages getting delayed and the corresponding timestamp becoming obsolete by the time messages reach the validators. In a real-world implementation of our system, this issue can be circumvented by allowing users to resubmit their transactions with new nonces if they get lost. Importantly, the DCN does not decrease the censorship resilience any further than the block-bound model does in comparison to the classical non-block-bound blockchain model, while having the advantage of providing guarantees against front-running, which neither the classical nor the block-bound model can achieve alone. This follows both under synchrony and asynchrony by the Honest-User Liveness property.

Permissioned Network. The DCN is designed as a permissioned network consisting of specialized parties offering efficient and reliable transaction timestamping. Importantly, the DCN only supplies transactions with timestamps and is therefore designed to be used together with an existing permissionless blockchain. In particular, the responsibility of adding blocks to the ledger, validating blocks and storing the blockchain itself remains in the hand of the permissionless set of miners or validators. Essentially, the permissioned nature of the DCN does not reduce the robustness and decentralization of the network of validators that verify the blocks, i.e., proves that they are honest. The permissionless network, thus, retains control of the most fundamental task.

Similar permissioned setups are already common in practice today. For instance, Chainlink oracles bringing price data from the real world onto the blockchain usually operate in a similar fashion [1]. Moreover, since Ethereum's transition from Proof-of-Work to Proof-of-Stake, block building has become more concentrated [24, 43, 45], in that currently more than 90% of the blocks are built with *proposer-builder separation (PBS)* [2]. In the first six months since the merge, a mere 133 builders have built these PBS blocks that were included on the ledger [24]. With PBS, block building is no longer done by the validators themselves but is instead handled by highly sophisticated block builders [2], similarly in spirit to how the DCN is used for timestamping transactions. By shifting tasks requiring a high degree of complexity away from validators, such as building blocks, or timestamping transactions in the case of the DCN, the requirements to run a validator node decrease. Consequently, in the long run, the number of validators is expected to increase, leading to a higher overall degree of decentralization of the consensus layer [2, 34], i.e., the core of the blockchain. In our case, parties participating in the DCN are now responsible for the non-trivial task of ordering transactions, while the complexity for the validators decreases. In particular, the task of block building becomes easier as validators must simply order transactions according to their timestamp.

Finally, we note that PBS and the DCN are incompatible. While the former optimizes for block value and thereby likely includes front-running transactions, the latter is designed to achieve a fair ordering that prevents front-running. If the DCN were integrated into a permissionless blockchain instead of PBS, the blockchain would protect users from front-running as opposed to maximizing block value on their behalf.

7 Conclusion and Future Work

We introduced the DCN, a novel and practical solution for fair transaction ordering in permissionless blockchains. Our approach differs from previous works by treating fair ordering as a Byzantine Agreement problem rather than a Byzantine State Machine Replication problem, leading to a simpler and faster algorithm while achieving good fairness guarantees. In particular, our new timestamp agreement protocol achieves $\lceil f/2 \rceil$-Median Fairness when the network is synchronous and falls back to a guarantee of f-Median Fairness during periods of asynchrony. These two bounds are the best that can be obtained in terms of δ-Median Fairness for the synchronous and asynchronous cases, respectively, as we have shown. The asynchronous fallback paradigm is a relatively unexplored, yet more robust notion than partial synchrony, so we find it natural to use it in designing other blockchain network protocols under realistic conditions.

As a next step, it would be valuable to consider the implementation of a dynamic set of nodes in the DCN, supporting the addition and removal of nodes in a controlled manner and the updating of related information. To do so, it will also be important to provide incentives for the nodes. One possible way to do so is to use rewards coming from transaction

fees, similar to gas fees in other blockchain systems. Additionally, it would be of interest to develop a prototype of our proposed method and evaluate its performance on-chain. Finally, the DeFi scene would benefit from the development of approaches for combating destructive front-running, which our work does not address.

References

1 Oracles – DefiLlama, 2023. URL: https://defillama.com/oracles.

2 Proposer-builder separation, 2023. URL: https://ethereum.org/nl/roadmap/pbs/.

3 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. doi:10.1007/11516798_17.

4 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.

5 Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. Sok: Mitigation of front-running in decentralized finance. *Cryptology ePrint Archive*, 2021. URL: https://eprint.iacr.org/2021/1628.

6 P. Berman and J.A. Garay. Randomized distributed agreement revisited. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 412–419, 1993. doi:10.1109/FTCS.1993.627344.

7 Erica Blum, John Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography Conference*, 2019. doi:10.1007/978-3-030-36030-6_6.

8 Erica Blum, Jonathan Katz, and Julian Loss. Tardigrade: An atomic broadcast protocol for arbitrary network conditions. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part II 27*, pages 547–572. Springer, 2021. doi:10.1007/978-3-030-92075-3_19.

9 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.

10 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *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 7*, pages 514–532. Springer, 2001. doi:10.1007/3-540-45682-1_30.

11 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

12 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 524–541, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-44647-8_31.

13 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132, 2000. doi:10.1145/343477.343531.

14 Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 316–333. Springer, 2022. doi:10.1007/978-3-031-18283-9_15.

15 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.

16 ChainLink Labs. Fair Sequencing Service (FSS), 2020. URL: https://blog.chain.link/chainlink-fair-sequencing-services-enabling-a-provably-fair-defi-ecosystem/.

17 Giovanni Deligios, Martin Hirt, and Chen-Da Liu-Zhang. Round-efficient byzantine agreement and multi-party computation with asynchronous fallback. In *Theory of Cryptography Conference*, pages 623–653. Springer, 2021. doi:10.1007/978-3-030-90459-3_21.

18 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.

19 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.

20 Flashbots. Flashbots. URL: https://docs.flashbots.net/.

21 Flashbots. Mev-explore. URL: https://explore.flashbots.net/.

22 R. Friedman, A. Mostefaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, 2005. doi:10.1109/TDSC.2005.13.

23 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.

24 Lioba Heimbach, Lucianna Kiffer, Christof Ferreira Torres, and Roger Wattenhofer. Ethereum's proposer-builder separation: Promises and realities. In *2023 ACM Internet Measurement Conference (IMC), Montreal, QC, Canada*, oct 2023. doi:10.1145/3618257.3624824.

25 Lioba Heimbach and Roger Wattenhofer. SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance. In *4th ACM Conference on Advances in Financial Technologies (AFT), Cambridge, Massachusetts, USA*, sep 2022. doi:10.1145/3558535.3559784.

26 Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. *Cryptology ePrint Archive*, 2021. URL: https://eprint.iacr.org/2021/1465.

27 Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*, pages 451–480. Springer, 2020. doi:10.1007/978-3-030-56877-1_16.

28 Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 25–36, 2020. doi:10.1145/3419614.3423263.

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

30 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.

31 Darya Melnyk and Roger Wattenhofer. Byzantine agreement with interval validity. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 251–260. IEEE, 2018. doi:10.1109/SRDS.2018.00036.

32 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with $t < n/3$, $O(N^2)$ messages, and $O(1)$ expected time. *J. ACM*, 62(4), sep 2015. doi:10.1145/2785953.

33 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

34 Pablo Pettinari. Proof-of-stake vs proof-of-work, 2023. URL: `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/pos-vs-pow/`.

35 Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 198–214. IEEE, 2022. `doi:10.1109/SP46214.2022.9833734`.

36 Michael O. Rabin. Randomized byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, SFCS '83, pages 403–409, USA, 1983. IEEE Computer Society. `doi:10.1109/SFCS.1983.48`.

37 Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, nov 1979. `doi:10.1145/359168.359176`.

38 Victor Shoup. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*, pages 207–220. Springer, 2000. `doi:10.1007/3-540-45539-6_15`.

39 David Stolz and Roger Wattenhofer. Byzantine agreement with median validity. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46, pages 22:1–22:14. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2015. `doi:10.4230/LIPICS.OPODIS.2015.22`.

40 David Stolz and Roger Wattenhofer. Byzantine Agreement with Median Validity. 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 1–14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPICS.OPODIS.2015.22`.

41 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`.

42 Nitin H Vaidya and Vijay K Garg. Byzantine vector consensus in complete graphs. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 65–73, 2013. `doi:10.1145/2484239.2484256`.

43 Anton Wahrstätter, Liyi Zhou, Kaihua Qin, Davor Svetinovic, and Arthur Gervais. Time to bribe: Measuring block construction market. *arXiv preprint*, 2023. `arXiv:2305.16468`.

44 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

45 Sen Yang, Fan Zhang, Ken Huang, Xi Chen, Youwei Yang, and Feng Zhu. Sok: Mev countermeasures: Theory and practice. *arXiv preprint*, 2022. `arXiv:2212.05111`.

46 Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. *Cryptology ePrint Archive*, 2020. URL: `https://eprint.iacr.org/2020/1300`.

Byzantine Consensus in Abstract MAC Layer

Lewis Tseng
Clark University, Worcester, MA, USA

Callie Sardina
University of California at Santa Barbara, CA, USA

Abstract

This paper studies the design of Byzantine consensus algorithms in an *asynchronous* single-hop network equipped with the "abstract MAC layer" [DISC09], which captures core properties of modern wireless MAC protocols. Newport [PODC14], Newport and Robinson [DISC18], and Tseng and Zhang [PODC22] study crash-tolerant consensus in the model. In our setting, a Byzantine faulty node may behave arbitrarily, but it cannot break the guarantees provided by the underlying abstract MAC layer. To our knowledge, we are the first to study Byzantine faults in this model.

We harness the power of the abstract MAC layer to develop a Byzantine approximate consensus algorithm and a Byzantine randomized binary consensus algorithm. Both of our algorithms require *only* the knowledge of the upper bound on the number of faulty nodes f, and do *not* require the knowledge of the number of nodes n. This demonstrates the "power" of the abstract MAC layer, as consensus algorithms in traditional message-passing models require the knowledge of *both* n and f. Additionally, we show that it is necessary to know f in order to reach consensus. Hence, from this perspective, our algorithms require the minimal knowledge.

The lack of knowledge of n brings the challenge of identifying a quorum explicitly, which is a common technique in traditional message-passing algorithms. A key technical novelty of our algorithms is to identify "implicit quorums" which have the necessary information for reaching consensus. The quorums are implicit because nodes do not know the identity of the quorums – such notion is only used in the analysis.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine, Randomized Consensus, Approximate Consensus, Abstract MAC

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.9

Related Version *Full Version*: `https://arxiv.org/abs/2311.03034`

Funding This material is based upon work partially supported by the National Science Foundation under Grant CNS-2238020. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

1 Introduction

We study the Byzantine consensus problems [21, 12, 27] in the "abstract MAC layer" model [20, 23, 17, 15]. The model was proposed by Kuhn, Lynch, and Newport [20] which harnesses the basic properties provided by existing wireless MAC (medium access control) protocols. The main purpose is to separate the high-level and low-level logic of algorithm design and the management of the wireless medium and participating nodes, respectively. Understanding the dynamics of these two levels helps one to explore the fundamental tradeoffs in algorithm design, and hopefully enables the development and deployment of high-level algorithms onto low-level MAC protocols [20, 26, 25].

The model is focused on an *asynchronous* single-hop network in which nodes communicate via "*mac-broadcasts*," the broadcast primitive provided by the abstract MAC layer. The mac-broadcast sends a message to all the fault-free nodes in the network, and the broadcaster

will eventually receive an acknowledgement (ACK) upon the successful completion of the mac-broadcast. That is, upon learning the ACK, the broadcaster can be sure that all the fault-free nodes have received the message that was broadcast. The abstract MAC layer additionally provides authentication of messages between unknown processes. This primitive is stronger than the traditional point-to-point message-passing [3, 22] in the sense that prior works have proposed *wait-free* crash-tolerant randomized consensus protocols [25, 26] in the abstract MAC layer, which is typically infeasible in the point-to-point message-passing models [3, 22].

To see the "power" of the abstract MAC layer, consider the case when node i sends a message m using mac-broadcast. Node i does *not* need to wait for explicit acknowledgement messages from other nodes. Instead, the abstract MAC layer provides the ACK which identifies the completion of the mac-broadcast. This indicates that all the other fault-free nodes have received the message m. In the case of traditional asynchronous point-to-point message-passing models, such guarantee is only ensured when node i receives acknowledgement messages from all the other fault-free nodes, which is impossible when nodes may crash.

Another important modeling choice is that prior works [20, 23, 17, 15, 25, 26, 31] assume little information known to the nodes to better capture the limitations of existing MAC protocols and the nature of the wireless networks. In particular, nodes do not have any a priori information about other nodes within the system. Some prior works (e.g., [25, 26]) even study anonymous algorithms in which nodes do not have unique identifiers. We assume that each node has a unique identifier, but does not have prior information on the size of the system, or the identifiers of any other node at the beginning of the algorithm.

Consensus in Abstract MAC Layer

To our knowledge, there are three prior papers [26, 25, 31] that study fault-tolerant consensus in the abstract MAC layer. All three works consider either no failure, or assume only crash faults. In [26], Newport proves several impossibilities and identify consensus algorithms when there is no failure. Newport and Robinson [25] propose two algorithms that employ the abstract MAC layer to solve randomized binary consensus when nodes may crash. Tseng and Zhang [31] provide another randomized consensus algorithm with improved storage complexity and expected time complexity. Approximate consensus is also studied in [31].

We are not aware of any work on tackling Byzantine faults in the abstract MAC layer. We assume that a Byzantine faulty node can behave arbitrarily. It may also send inconsistent messages to other nodes. The only constraint of the Byzantine adversary is that it *cannot* break the underlying abstract MAC layer. First, it cannot "nullify" the delivery guarantee provided by the ACK at fault-free nodes. Second, even though nodes do not have information about other nodes a priori, communication is authenticated (by the MAC layer), and the receiver can verify the identity of the message sender, once it receives the message. Consequently, a Byzantine adversary cannot fake its identity. From a more practical perspective, we are focused on the Byzantine faults at the application layer. For example, the Byzantine adversary under our consideration cannot use jamming or sybil attacks to undermine the abstract MAC layer.

The Fischer-Lynch-Paterson (FLP) impossibility result [13] proves that it is impossible to design a deterministic exact consensus algorithm when nodes may fail in asynchronous message-passing systems. The result can be extended to the abstract MAC layer model [26]. Therefore, we focus on approximate consensus [12] and randomized binary consensus [27] problems. In the first problem, the agreement property is relaxed so that the outputs at fault-free nodes only need to be roughly equal, whereas in the second problem, the termination only holds in a probabilistic sense.

Our Contributions

Consider an asynchronous system consisting of n nodes and up to f Byzantine faulty nodes. We propose the two following Byzantine-tolerant consensus algorithms:

- *Approximate Consensus*: We present MAC-BAC, which is correct given $n \geq 5f + 2$. Similar to prior approximate algorithms [12, 1], nodes proceed in rounds and maintain a state value which is updated every round and eventually will become the output. MAC-BAC achieves convergence rate 3/4. More concretely, after every two rounds, the range of the state values at fault-free nodes is reduced by at least 1/4.
- *Randomized Consensus*: We present MAC-RBC, which is correct given $n \geq 5f + 1$. The expected time complexity is constant. The algorithm assumes the existence of a common coin [27] among all fault-free nodes.

Our algorithms require *only* the knowledge of f, and do *not* need to know n. We also prove that the knowledge of f is necessary for solving Byzantine consensus in [30].

Our model is weaker than the synchronous point-to-point message-passing model [21]; hence, the lower bound on resilience $3f + 1$ still applies. Moreover, a node can use mac-broadcast to simulate a point-to-point communication if n is known. Therefore, prior algorithms [1, 27, 24] with optimal resilience $3f + 1$ can be simulated in our model with the knowledge of n. The lower bound on resilience when n is unknown is left as an interesting future work.

The lack of knowledge of n makes it impossible to identify a quorum explicitly, which is a common technique in consensus algorithms in the traditional message-passing models, e.g., [3, 22, 21, 12, 27, 24]. A key novelty of our algorithms is to identify "implicit quorum." More precisely, our technique ensures that there exists a quorum whose information will eventually be propagated to other fault-free nodes; however, nodes do not know the explicit identifiers of the nodes inside the quorum.

One challenge of such an implicit quorum is that the analysis becomes more complicated, as we first need to ensure that an implicit quorum exists and then we need to argue that other nodes will be able to learn the necessary information from the implicit quorum (which is not always obvious due to asynchrony and Byzantine faults).

2 Related Work

We discuss the most relevant consensus algorithms and the works on the abstract MAC layer. Modeling wireless networks with the abstract MAC layer was first introduced by Kuhn, Lynch, Newport in [20], in which they present algorithms for multi-message broadcasts in a multi-hop network when there is no failure. Non-fault-tolerant leader election and maximal independent set problems are later studied in the model [23, 17, 15].

The three prior works [26, 25, 31] that study fault-tolerance in the abstract MAC layer all focus on crash faults. The techniques are different from our work, because Byzantine adversary can send inconsistent messages. For example, a technique of "counter racing" (for identifying when to output a value safely) is used in [25] and a technique of "jumping" to a state proposed by another node is proposed in [31]. These techniques do not work if a Byzantine node lies about its observations or state value.

The problem of Byzantine consensus in message-passing has been extensively studied in the literature since the seminal work by Lamport, Shostak and Pease [21]. Dolev et al. [12] propose an iterative approximate Byzantine consensus algorithm that is correct given $n \geq 5f + 1$. Our algorithm MAC-BAC is inspired by their algorithm and requires $n \geq 5f + 2$. Mostefaoui et al. [24] propose a Byzantine randomized binary consensus with

optimal resilience $n \geq 3f + 1$ and achieve expected constant time complexity. Our MAC-RBC algorithm is inspired by their algorithm, but requires $n \geq 5f + 1$. Both algorithms use "common coin" [27], which guarantees that every node receives the same sequence of random bits. Unlike prior algorithms, MAC-BAC and MAC-RBC does not use the information regarding n or the notion of "explicit quorum," so our design and analysis are more complicated than the ones in [12, 24].

Abrahm et al. [1] present an approximate consensus algorithm with optimal resilience. Their algorithm relies on the reliable broadcast primitive and the witness technique. Many works [6, 28, 29] describe Byzantine randomized binary consensus algorithms with various guarantees. These algorithms all achieve optimal resilience $n \geq 3f + 1$. There are also recent works on Byzantine randomized consensus that require more powerful primitives such as PKI [10, 14, 5, 9, 7]. All these algorithms require the knowledge of n and rely on the usage of explicit quorum and some variation of reliable broadcasts [6].

Without knowing n, it is difficult to identify quorums explicitly so that there is an intersection between any two quorums. For example, in many prior works that use reliable broadcast (e.g., [1, 6]), a quorum of size $n - f$ is used, which ensures that a Byzantine node cannot equivocate. However, when n is unknown, it is unclear whether such property can be guaranteed, forcing us to develop new techniques. In fact, the lower bound on the resilience of Byzantine consensus problems in the abstract MAC layer is still an open problem.

There is also a line of works aiming to reach consensus in synchronous systems with unknown participants. The problem is named CUPs (Consensus with Unknown Participants). Similar to our model, the CUPs problem assumes no knowledge of n. It was first studied by Cavin et al. [8] when nodes do not crash. Greve and Tixeuil [16] study the tradeoffs between synchrony and the shared knowledge between nodes in a multi-hop network. Later, Alcheriri et al. [2] and Khanchandani and Wattenhofer [18] consider the Byzantine consensus in CUPs. These work assume synchrony; hence, are very different from our model.

3 Preliminaries

3.1 System Model

Our system model consists of a static system with n nodes, with up to f nodes which may be Byzantine faulty. The set of nodes is denoted as the set of their unique identifiers, i.e., $\{1, \ldots, n\}$. However, the knowledge of n is only used in analysis. In our algorithms, nodes do not know n. Moreover, due to asynchrony and faults, it is impossible to learn n exactly.

Byzantine nodes may send arbitrary messages to other nodes, or act as crashed nodes. The messages which Byzantine nodes send to all other nodes need not be consistent. We assume that the behavior of the Byzantine nodes is controlled by a malicious adversary with access to the system state throughout the algorithm. Nodes which are not Byzantine are called fault-free nodes. Fault-free nodes follow the algorithm protocol. Our algorithm MAC-BAC assumes $n \geq 5f + 2$, and MAC-RBC assumes $n \geq 5f + 1$.

Our algorithm operates on top of a single-hop network equipped with the abstract MAC layer [19]. The model provides a communication primitive "mac-broadcast," which ensures an eventual delivery guarantee. More specifically, at some point after a node i has broadcast a message via "mac-broadcast," node i will receive an acknowledgment (ACK) which indicates that all other fault-free nodes within the system have received i's message. No other information is contained within the ACK, e.g., the ACK relays no information concerning the number of other nodes within the system. As discussed in Section 1, we consider Byzantine faults in the application layer; hence, the guarantees of the underlying abstract MAC layer cannot be disrupted by the Byzantine adversary.

3.2 Approximate and Randomized Consensus

A correct approximate consensus algorithm [12] must satisfy the following conditions:

- *Termination*: Every fault-free node must output a value in a finite amount of time.
- *Validity*: the output must remain in the convex hull of the inputs of the fault-free nodes.
- ϵ-*Agreement*: For any $\epsilon > 0$, the output of all fault-free nodes are within ϵ of each other.

A correct randomized binary consensus algorithm [27, 4] must satisfy the following conditions when the input is a binary value (either 0 or 1):

- *BC-Termination*: Every fault-free node outputs a value with probability 1.
- *BC-Validity*: Every output value was proposed by a fault-free node.
- *BC-Agreement*: The output of all fault-free nodes are identical.

4 Byzantine Approximate Consensus: MAC-BAC

This section presents our algorithm MAC-BAC, which is a correct Byzantine approximate consensus given $n \geq 5f + 2$. It follows the structure of the algorithm by Dolev et al. [12]. In both algorithms, node i proceeds in rounds and keeps a state value v_i that eventually becomes the output, after a sufficient number of rounds. The key difference between the two algorithms is that in MAC-BAC, node i waits until it receives at least $4f + 2$ messages from the same round (instead of $n - f$ in [12]). By assumption, node i is able to transmit a message to itself using mac-broadcast.

Recall that in our model, we assume nodes do not have the knowledge of n. Consequently, we do not have the notion of explicit quorum. (In [12], the $n - f$ nodes from which a node i received a message act as a quorum.) Therefore, our analysis is more complicated in the sense that we need to identify how important information is propagated throughout the rounds, via the help of "implicit quorum."

4.1 MAC-BAC

MAC-BAC is presented in Algorithm 1. The first step of the algorithm is to broadcast its identifier, its current value and round index using mac-broadcast. Once this mac-broadcast has completed, an ACK will be received from the abstract MAC layer acknowledging that the message has been received by all the fault-free nodes.

Each node i then waits to receive at least $4f + 2$ messages from round p_i. Upon receiving these messages, node i discards extreme values and update its new state value. We introduce two notations to facilitate the presentation:

- $\min^{f+1}\{R_i[p_i]\}$ denotes the $(f + 1)$-st minimum value in $R_i[p_i]$;[1] and
- $\max^{f+1}\{R_i[p_i]\}$ denotes $(f + 1)$-st maximum value in $R_i[p_i]$.[2]

Our strategy of updating the state value is as follows: at line 5, l takes the $(f + 1)$-st minimum value in $R_i[p_i]$. At line 6, u takes the $(f + 1)$-st maximum value in $R_i[p_i]$. The new state value at node i is then updated to be the average of l and u, at line 7. This is also the strategy used in [1, 11].

Node i then proceeds to the next round. Once node i reaches the final round, p_{end}, it outputs the final state value, $v_i[p_{end} + 1]$.

[1] Alternatively, the smallest value after discarding f smallest values in $R_i[p_i]$.
[2] Alternatively, the largest value after discarding f largest values in $R_i[p_i]$.

Algorithm 1 MAC-BAC: Steps at each node i.

Local Variables:

p_i ▷round index, initialized to 0
v_i ▷state, initialized to x_i, the input at node i

1: **for** $p_i \leftarrow 0$ to p_{end} **do**
2: **mac-broadcast**(i, v_i, p_i)
3: **wait until** node i has received $\geq 4f + 2$ messages from round p_i
4: $R_i[p_i] \leftarrow$ received round-p_i messages
5: $l = \min^{f+1}\{R_i[p_i]\}$
6: $u = \max^{f+1}\{R_i[p_i]\}$
7: $v_i[p_i + 1] \leftarrow \frac{l+u}{2}$
8: $p_i \leftarrow p_i + 1$
9: **end for**
10: **output** $v_i[p_{end} + 1]$

4.2 Correctness Proof

Termination is obvious, as p_{end} is a fixed value defined in Eq. (1). Moreover, since there are at least $5f + 2$ nodes, each node is able to receive enough messages at Line 3. We present the proof in [30]. Validity also follows from the strategy of discarding extreme values. Essentially, both l and u are guaranteed to be in the convex hull of the state values (v_i's) at fault-free nodes from the previous round. The proof is presented in [30].

A key novelty is the way we prove ϵ-agreement. In prior works [1, 12], the range of state values at fault-free nodes shrinks every round, whereas in our proof, the range shrinks every *two* rounds. Moreover, in prior algorithms, any pair of two fault-free nodes must use at least one identical value to update their new state values, due to the usage of an explicit quorum. However, such a condition might not hold for MAC-BAC. This is because n is unknown, and nodes might receive messages from two groups of nodes such that the intersection of the two group is less than f nodes. In this case, there is no guarantee that nodes will use common value(s) to update the state values. In fact, in our algorithm, some nodes might use completely different values for updating (i.e., after discarding the common values) in the same round.

4.2.1 Proof of ϵ-Agreement and Implicit Quorum in MAC-BAC

Useful Notions

We first introduce some terminology to facilitate the proof.

▶ **Definition 1** (First and Second Mover). *For each round r, the set of first movers is defined as the first $2f + 1$ fault-free nodes that complete their respective mac-broadcasts (at Line 2) in round r.*[3] *All the other fault-free nodes are called second movers.*

In our analysis, we are interested in how first and second movers propagate and update their values. Therefore, we introduce two sets F_r and S_r below.

▶ **Definition 2.** *Let F_r be the set of state values of the first movers at the end of round r – the v_i **after** a first mover i updates its state value at Line 7 in round r. Let S_r be the set of state values of the second movers at the end of round r – the v_j **after** a second mover j updates its state value at Line 7 in round r.*

▶ **Observation 3** (Sequential Order). *Without loss of generality, we can assume nodes complete Line 2 following a sequential order for each round.*

[3] We can break ties using IDs without affecting the correctness.

For brevity of the presentation, we relabel the IDs so that node j completes before node i.[4]

We know by sequential ordering that if $j < i$, then j completes its mac-broadcast before i. This means that node j must have received its ACK from the mac-broadcast before node i completes Line 2. Therefore, in order to move to the next round $r+1$, i must receive node j's round-r state, i.e., $v_j[r]$ that is assigned at Line 5. Wth a slight abuse of terminology, let node's round-0 state be the input for that node.

▶ **Observation 4.** *Following the sequential order and the property of the mac-broadcast, we know that for each round $r+1$, node i must receive node j's round-r state if $j < i$ for all fault-free i and j.*

Note that by definition, if j is fault-free, then it is either a first or second mover. Additionally, this observation does not indicate the relationship between $R_i[r+1]$ and $R_j[r+1]$. In particular, it is possible that $v_k[r] \in R_i[r+1]$ and $v_k[r] \notin R_j[r+1]$. This is possible if $i, j < k$ or $j < k < i$.

Observation 4 and the guarantees of the abstract MAC layer together imply that the state values broadcast by the first movers are received by all the second movers.

Implicit Quorum in MAC-BAC

In our analysis, first movers are the "implicit quorum" for second movers in round r, due to Observation 4. This is because even though second movers do not know the identities of the first movers, second movers will share the same information from the first movers and use some of the state values at first movers to update their new state values.

Interestingly, first movers may not have enough shared information in round r *within themselves*. This is possible if they receive many messages from non-overlapping sets of second movers at Line 3. They are only guaranteed to receive common information from their "implicit quorum" in the next round. More concretely, first movers of round $r+1$ are guaranteed to receive enough information (for convergence) from the second movers of round r. This is because node i waits for $4f+2$ messages. Among them, $2f+1$ could be from first movers of round r, f could be from Byzantine nodes, and the remaining $f+1$ could be from second movers of round r. This turns out is enough for first movers of round $r+1$ to converge. The proof of Lemma 7 presents this intuition in more detail.

Proof of ϵ-Agreement

Without loss of generality, we can scale the inputs to $[0,1]$ as long as we scale ϵ down by the same factor. For simplicity of the presentation, we assume that for each fault-free node i, its input $x_i \in [0,1]$.

We first prove the following lemma. The lemma below follows from the fact that each node discards extreme values from Byzantine nodes. The proof is presented in [30].

▶ **Lemma 5.** *Fix a round $r \geq 1$. Assume the range of state values at fault-free nodes is $[x,y]$, where $0 \leq x, y \leq 1$, i.e., $F_r \cup S_r = [x,y]$. Then, we have $F_{r+1} \subseteq F_r \cup S_r = [x,y]$.*

[4] Assuming that nodes complete Line 2 in a sequential order within each round does not affect the correctness because nodes only process messages received from nodes at the same round, and the state of a node changes only once within a round. When the state updates at Line 7, the round number increments as well on Line 8 (only round p messages are processed in round p, so any change of states is the value for the subsequent round). Ordering does not alter the values sent to/ from nodes, nor does it allow for values to be considered from the incorrect round. Therefore, correctness is not violated, after the ID relabeling.

Using the same argument, we can also show that $F_1 \subseteq [0, 1]$, the range of input x_i.

Before proving how the state values at second movers evolve, we first introduce two notations for a round $r \geq 1$:

- Let m_r be the minimum fault-free state value at the end of round r, $m_r = \min\{F_r \cup S_r\}$.
- Let M_r be the maximum fault-free state value at the end of round r, $M_r = \max\{F_r \cup S_r\}$.

It follows that the interval length of $F_r \cup S_r$ is $M_r - m_r$.

Then, we prove the following lemma. The lemma is where we utilize the "implicit quorum" for second movers. By the property of mac-broadcast, every second mover must receive the state values from all the first movers (Observation 4); hence, we can use this observation to show that the interval length of S_{r+1} must shrink by at least half. This particular proof is similar to the ones in traditional message-passing networks [12, 1].

▶ **Lemma 6.** *Fix round $r \geq 1$. The interval length of S_{r+1} is at most half of the interval length of $F_r \cup S_r$.*

Proof. Let x_{r+1} be the median of the state values at first movers in round $r + 1$. By definition, we have $m_r \leq x_{r+1}$ and $M_r \geq x_{r+1}$.

Now, consider any two second movers i and j. Without loss of generalization, assume $v_i[r + 2] \geq v_j[r + 2]$. Recall that these values are produced at the end of round $r + 1$ at Line 7.

Since i discards extreme values, $u \leq M_r$ and $l \leq x_{r+1}$ at node i. Similarly, $x_{r+1} \leq u$ and $m_r \leq l$ at node j. Therefore, we have in round $r + 1$,

$$v_i[r + 2] = \frac{l + u}{2} \leq \frac{x_{r+1} + M_r}{2}$$

and

$$v_j[r + 2] = \frac{l + u}{2} \geq \frac{m_r + x_{r+1}}{2}$$

Consequently, we have

$$v_i[r + 2] - v_j[r + 2] \leq \frac{M_r + x_{r+1}}{2} - \frac{x_{r+1} + m_r}{2} = \frac{M_r - m_r}{2}$$

Since the inequality applies to any pair of second movers i and j, the interval length of S_{r+1} is at most half of the interval length of $F_r \cup S_r$. (Note that the interval length of $F_r \cup S_r$ is simply $M_r - m_r$.) ◀

The proof can be easily applied to the case of S_1. That is, the interval length of S_1 is at most half of the interval length of the inputs, $[0, 1]$.

We then prove the following key lemma. This proof is where we use the notion of implicit quorum for first movers. In particular, first movers in round $r + 1$ rely on second movers in round r to learn the information that is essential for convergence.

▶ **Lemma 7.** *The interval length of $F_{r+2} \cup S_{r+2}$ is at most $\frac{3}{4}(M_r - m_r)$.*

Proof. First, let us define

- $F_{r+1} = [a^F_{r+1}, b^F_{r+1}]$
- $S_{r+1} = [a^S_{r+1}, b^S_{r+1}]$

Note that all these four values (at a respective bound) are in the range of $[m_r, M_r]$ due to Lemma 5.

Now, we consider the smallest possible value for $F_{r+2} \cup S_{r+2}$.

- Case I: if $a^F_{r+1} < a^S_{r+1}$.

  In this case, the smallest value is $\frac{a^F_{r+1}+a^S_{r+1}}{2}$. This is because there are at most $2f+1$ a^F_{r+1} in F_{r+1} and up to f Byzantine nodes can send values $\leq a^F_{r+1}$. The remaining $f+1$ values must come from S_{r+1} whose smallest value is a^S_{r+1}. After discarding f values, at least a value that is $\geq a^S_{r+1}$ remains to be used to update the state value at Line 7.
- Case II: if $a^F_{r+1} \geq a^S_{r+1}$.

  In this case, the smallest value is a^S_{r+1}. This is because there could be more than $4f+2$ a^S_{r+1}'s in $S_{r+1} \cup F_{r+1}$.

Next, we consider the largest possible value for $F_{r+2} \cup S_{r+2}$.

- Case III: if $b^F_{r+1} > b^S_{r+1}$.

  In this case, the largest value is $\frac{b^F_{r+1}+b^S_{r+1}}{2}$. This is because there are at most $2f+1$ b^F_{r+1} in F_{r+1} and up to f Byzantine nodes can send values $\geq b^F_{r+1}$. The remaining $f+1$ values must come from S_{r+1} whose largest value is b^S_{r+1}. After discarding f values, at least a value that is $\leq b^S_{r+1}$ remains to be used to update the state value at Line 7.
- Case IV: if $b^F_{r+1} \leq b^S_{r+1}$.

  In this case, the largest value is b^S_{r+1}. This is because there could be more than $4f+2$ b^S_{r+1}'s in S_{r+1}.

Now, we can consider the following four cases to bound the interval length of $F_{r+2} \cup S_{r+2}$:

- $a^F_{r+1} < a^S_{r+1}$ and $b^F_{r+1} > b^S_{r+1}$:
  The interval length is

$$\begin{aligned}\frac{b^F_{r+1}+b^S_{r+1}}{2} - \frac{a^F_{r+1}+a^S_{r+1}}{2} &= \frac{1}{2}\{(b^F_{r+1}-a^F_{r+1}) + (b^S_{r+1}-a^S_{r+1})\} \\ &= \frac{1}{2}\{(M_r-m_r) + (M_r-m_r)/2\} = \frac{3(M_r-m_r)}{4}\end{aligned}$$

- $a^F_{r+1} < a^S_{r+1}$ and $b^F_{r+1} \leq b^S_{r+1}$:
  The interval length is

$$\begin{aligned}b^S_{r+1} - \frac{a^F_{r+1}+a^S_{r+1}}{2} &= \frac{1}{2}\{(b^S_{r+1}-a^S_{r+1}) + (b^S_{r+1}-a^F_{r+1})\} \\ &= \frac{1}{2}\{(M_r-m_r)/2 + (M_r-m_r)\} = \frac{3(M_r-m_r)}{4}\end{aligned}$$

- $a^F_{r+1} \geq a^S_{r+1}$ and $b^F_{r+1} > b^S_{r+1}$:
  The interval length is

$$\begin{aligned}\frac{b^F_{r+1}+b^S_{r+1}}{2} - a^S_{r+1} &= \frac{1}{2}\{(b^F_{r+1}-a^S_{r+1}) + (b^S_{r+1}-a^S_{r+1})\} \\ &= \frac{1}{2}\{(M_r-m_r) + (M_r-m_r)/2\} = \frac{3(M_r-m_r)}{4}\end{aligned}$$

- $a^F_{r+1} \geq a^S_{r+1}$ and $b^F_{r+1} \leq b^S_{r+1}$:
  The interval length is

$$b^S_{r+1} - a^S_{r+1} = \frac{(M_r-m_r)}{2}$$

In each case above, we saw that compared to $F_r \cup S_r$, the interval length of $F_{r+2} \cup S_{r+2}$ shrinks by at least $1/4$, proving the lemma. ◀

Now, we are ready to prove that MAC-BAC converges with the desirable convergence rate.

▶ **Theorem 8.** *MAC-BAC achieves ϵ-agreement in*

$$p_{end} \geq 2 \cdot \log_{\frac{3}{4}} \epsilon \tag{1}$$

rounds.

Proof. By the conclusion of Lemma 7, $F_r \cup S_r \leq \frac{3}{4} \cdot F_{r-2} \cup S_{r-2}$. Therefore, to satisfy ϵ-agreement, the number of iteration r must satisfy the following inequality.

$$\begin{aligned}
\epsilon &\geq \frac{3}{4}^{\lfloor \frac{r}{2} \rfloor} \\
\log \epsilon &\geq \lfloor \frac{r}{2} \rfloor \cdot \log \frac{3}{4} \\
2 \cdot \log \epsilon &\geq r \cdot \log \frac{3}{4} \\
\frac{2 \cdot \log \epsilon}{\log \frac{3}{4}} &\leq r \\
r &\geq 2 \cdot \log_{\frac{3}{4}} \epsilon
\end{aligned}$$

Therefore, ϵ-agreement will be achieved in $\geq 2 \cdot \log_{\frac{3}{4}} \epsilon$ rounds. If we define p_{end} as the smallest integer that satisfies the inequality in Algorithm MAC-BAC, then ϵ-agreement is achieved. ◀

5 Byzantine Randomized Binary Consensus: MAC-BRC

Assuming $n \geq 5f + 1$, our algorithm MAC-BRC correctly solves Byzantine randomized binary consensus. In MAC-BRC, each node is assumed to have a common coin provided by a trusted dealer, as in the work by Rabin [27], which guarantees that every node shares the same sequence of random bits $b_1, b_2, \cdots, b_k, \cdots$ with value 0 or 1, each with probability $\frac{1}{2}$. Additionally, the common coin is "global," meaning that the k-th call to $coinflip()$ (Line 9 of Algorithm MAC-RBC) by a fault-free node will return the same bit, b_c, to all nodes invoking the k-th $coinflip()$.

Our algorithm is inspired by [24], especially the way we use the common coin to decide whether it is safe to output a value. As mentioned earlier, the key technical contribution is the usage of implicit quorum, which will become clear when we present the algorithm.

5.1 MAC-RBC

MAC-RBC is presented in Algorithm 2. Nodes proceed in phases. In phase p, each node i first does a mac-broadcast of an EST message containing i's "estimated" value (that i estimates to be the output based on the information it collected in the previous phase) and phase number to other nodes. Once this mac-broadcast has completed, an ACK will be received. Meanwhile, a background event handler processes all EST messages to determine when a value can be safely added to a local estimated set, $estValues_i[p]$. The way the handler is constructed ensures that the value added $estValues_i[p]$ must be an estimate value v_i by some fault-free node.

Once $estValues_i[p]$ is non-empty, the main thread resumes at Line 4, where node i mac-broadcasts an AUX message for this value at node i. Another background event handler processes AUX messages, and adds the identifiers of all nodes which sent AUX messages

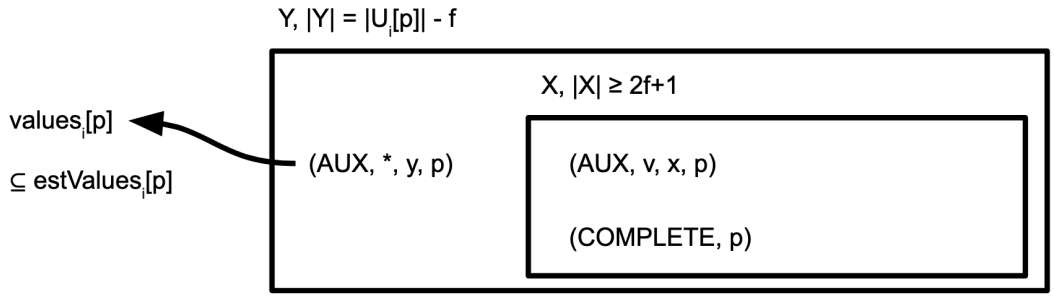


Figure 1 Illustration of Condition WAIT.

(for a particular value w) to its set U_i. U_i is used to count the number of other nodes that supports a certain value w. Node i then mac-broadcasts a $COMPLETE$ message indicating that it has completed its broadcasting of an AUX message.

Once our Condition WAIT (defined below) – which ensures that a node only adds fault-free values which have been sent and received by sufficiently many other nodes – is satisfied, node i will have some value(s) in its set $values_i[p]$. A call to $coinflip()$ on Line 9 employs the global common coin whose returned value c is then compared to $values_i[p]$. If these values are equal, that is $values_i[p] = \{v\} = c$, node i will output v. Otherwise, node i will adopt the value of the common coin and continue to the subsequent phase until it outputs a value.

The key novelty is the construction of Condition WAIT, defined in Definition 9. The condition makes sure that enough information is shared between any pair of fault-free nodes upon the satisfaction of Condition WAIT. On a high-level, the condition relies on two key elements: (i) a counter set $U_i[p]$ that keeps track of nodes that i knows; and (ii) an "implicit quorum" X which contains the nodes that saw the same estimate value v and have completed their mac-broadcast. Note that X is implicit in the sense that the X at node i might not always intersect with the X at node j. However, it turns out that it is already sufficient for our purpose. (More details in the proof of BC-Agreement in Section 5.2.1.)

▶ **Definition 9** (Condition WAIT). *In phase p, node i satisfies Condition WAIT with value v, if there exist two sets of nodes X and Y such that*

1. $|X| \geq 2f + 1$;
2. *i received $(COMPLETE, p)$ message for each $x \in X$;*
3. *i received (AUX, v, x, p) message for each $x \in X$ and some identical value v;*
4. $|Y| = |U_i[p]| - f$;
5. *i received $(AUX, *, y, p)$ message for each $y \in Y$;*[5]
6. $X \subseteq Y$;
7. *let $value_i[p]$ be the set of values contained in Y's $(AUX, *, *, p)$ messages;*
8. $value_i[p] \subseteq estValues_i[p]$.[6]

Figure 1 illustrates the relation between set Y and set X specified in Condition WAIT.

[5] Note that these AUX messages might not contain v.
[6] Note that $estValues_i[p]$ could keep growing even after the execution of Line 3 to Line 5, as the background message handler is long-living.

Algorithm 2 MAC-RBC: Steps at each node i.

Local Variables:

p_i ▷phase, initialized to 0
v_i ▷state, initialized to x_i, the input at node i
$estValues_i[p]$ ▷set, initialized to {}
$U_i[p]$ ▷set, initialized to {}

1: **while** true **do**
2: **mac-broadcast**(EST, v_i, p_i)
3: **wait until** $estValues_i[p_i] \neq \emptyset$
4: **for** each $w \in estValues_i[p_i]$ **do**
5: **mac-broadcast**(AUX, w, i, p_i)
6: **end for**
7: **mac-broadcast**$(COMPLETE, p_i)$
8: **wait until** Condition WAIT is satisfied with some value z
9: $c \leftarrow coinflip()$
10: **if** $values_i[p_i] = \{v\}$ **then**
11: **if** $v = c$ **then**
12: output v
13: **end if**
14: $v_i \leftarrow v$
15: **else**
16: $v_i \leftarrow c$
17: **end if**
18: $p_i \leftarrow p_i + 1$
19: **end while**

//Background EST message handler
Upon receiving (EST, v, p) **do**
20: **if** (EST, v, p) is received from $f + 1$ nodes and (EST, v, p) not yet broadcast by i **then**
21: **mac-broadcast**(EST, v, p))
22: **end if**
23: **if** (EST, v, p) is received from $2f + 1$ nodes **then**
24: $estValues_i[p] \leftarrow estValues_i[p] \cup \{v\}$
25: **end if**

//Background AUX message handler
Upon receiving $(AUX, *, j, p)$ **do**
26: $U_i[p] \leftarrow U_i[p] \cup \{j\}$ ▷Even if j sends two different AUX msgs, j is added only once

5.2 Correctness Proof

The BC-Validity proof follows from the construction of $estValues_i[p]$ (the $f + 1$ threshold), and the observation that the output must be some value from $estValues_i[p]$.

▶ **Theorem 10.** *Given that $n \geq 5f + 1$, MAC-RBC satisfies BC-Validity.*

Proof. Fix a phase p and let node i be a fault-free node with value $v \in values_i$ which has been mac-broadcast as an estimate value by a fault-free node. By the wait statement at Line 3, since each fault-free node i mac-broadcasts the values within its set $estValues_i$, and by the Condition WAIT, the set $values_i$ contains only values from fault-free nodes. The set $estValues_i$ contains only values from fault-free nodes because all values added to $estValues_i$ must have been sent by at least $f + 1$ nodes on in order to pass Line 20. There are at most f Byzantine nodes, so one of these (EST, v, p) messages must have been broadcast by a fault-free node.

If $values_i = \{v\} = c$, the value of the common coin, node i outputs v at Line 13 and sets its estimate value to c. If $values_i = \{v, v'\}$, both values have been processed by fault-free nodes, and node i adopts the value of the common coin as the estimate value for node i in phase $p_i + 1$ at Line 17. In both cases, the estimate value of a fault-free node is a value that has been proposed by a fault-free node. ◀

The BC-Termination proof is focused on showing that as long as $n \geq 5f+1$, then Condition WAIT can always be satisfied under all possible scenarios. Moreover, the termination with probability 1 roughly follows the proof structure in [24], which relies on the usage of common coin and the cardinality of $value_i[p]$ (the condition to check at Line 10). The full proof is presented in [30].

▶ **Theorem 11.** *Given that $n \geq 5f + 1$, MAC-RBC terminates with probability 1.*

5.2.1 Proof of BC-Agreement and Implicit Quorum in MAC-RBC

Let us define $values_i^r[p]$ to be the set of values $values_i[p]$ right after node i completes Line 8. That is, the $values_i[p]$ that node i derived from Condition WAIT (Definition 9). As mentioned earlier, X identified in Condition WAIT could be different for two different nodes. This is because n is unknown, and two nodes might use different sets of $2f+1$ nodes as X. However, in the proof of lemma below, we demonstrate that under a certain case, X at node i is guaranteed to intersect with Y at node j. This is mainly the usage of the COMPLETE message. Even though this claim does *not* imply that X at node i will intersect with X at node j; however, due to the usage of a common coin, this is already enough for showing agreement of MAC-RBC.

▶ **Lemma 12.** *Fix a phase p. For any fault-free i and j with $values_i^r[p] = \{v\}$ and $values_j^r[p] = \{u\}$, then $v = u$.*

Proof. Assume node i completes line 8 at time T_i with $values_i^r[p] = \{v\}$. By construction, node i has Condition WAIT satisfied with value v.

From conditions (1), (4), (6) and (7) of Condition WAIT, node i has received AUX messages from a set Y_i of size $|U_i[p]| - f \geq 2f+1$. (At this point, $estValues_i[p]$ might contain some value other than v, but we do not care about it.) We first prove the following claim.

▷ Claim 13. At least $f+1$ fault-free nodes in Y_i have completed line 5 with value v in the AUX message before time T_i. That is, at least $f+1$ fault-free nodes have mac-broadcast(AUX, v, y, p) for some $y \in Y_i$ by time T_i.

Proof of Claim 13. By condition (3) and (6) in Condition WAIT, Y_i contains X_i, and every fault-free node in X_i have completed line 5 with value v before broadcasting the $COMPLETE$ message. Since $|X_i| \geq 2f+1$, and up to f nodes can be Byzantine, at least $f+1$ nodes in X_i has completed line 5 with value v before time T_i, proving the claim. ◁

Let us denote the set of fault-free nodes identified in Claim 13 by Y^v. (Note that Y_v is a superset of X_i)

Now consider node j with $values_j^r[p] = \{u\}$. Without loss of generality, assume that j completes line 8 at some later time T_j, i.e., $T_j \geq T_i$. This assumption together with Claim 13 imply that $U_j[p]$ contains Y^v at time T_j, due to the guarantee of mac-broadcast.

By the definition of Condition WAIT, Y_j contain $|U_j[p]| - f$ nodes at time T_j. This implies that the intersection of Y_j and Y^v is non-empty. This is because the size of Y^v is at least $f+1$. Therefore, value v must be in $value_j[p]$, i.e., $v \in value_j[p_j]$. Since $value_j[p]$ contains a single element, this implies that $v = u$. ◀

▶ **Theorem 14.** *Given $n \geq 5f+1$, MAC-RBC satisfies BC-Agreement.*

Proof. Let phase p be the first phase at which a fault-free node i outputs a value v at Line 12. For any j that also outputs a value in phase p, both i and j must output the same value, namely, the value of the common coin.

Consider any node j that has not output in phase p. Observe that we have $values_i[p] = \{v\}$. It is then impossible by Lemma 12 for $values_j[p] = \{v'\}$, with $v' \neq v$. Therefore, $values_j = \{v, v'\}$. Node j will then execute Line 16 and set v_j to be the value of the common coin in phase p. By construction, this value is v.

Then, node j's estimate value in phase $p+1$ will be v. The output values must be an estimate value, and in phase $p+1$, all fault-free nodes have the same estimate value and will hence output v. This proves that the BC-Agreement property of MAC-RBC. ◀

6 Impossibility

In this section, we provide the intuition for our proof that without the knowledge of f, it is impossible to solve consensus. The full proof is included in [30].

We construct an indistinguishably proof by constructing scenarios in which there is no way for a node i to distinguish whether another node j's behavior is Byzantine or not, which could lead to a violation of validity. We assume the existence of an algorithm A which solves consensus for a certain n and f without the knowledge of these values. We construct two scenarios with different value for n and f. These scenarios remain indistinguishable to all nodes because no nodes have knowledge of n or f. In the first scenario, node i can only communicate with one other node to update its state, and in the second, node i can receive messages from $\geq 2f + 1$ other nodes within the system. In the second scenario, we impose a time delay, D, on all messages sent from any node other than an arbitrary node j. We observe that within the time interval $(0, D]$, a node i cannot distinguish between the two scenarios (whether there is only one, or more than one other nodes within the system). Now, node i has no way of determining whether the behavior of node j is Byzantine or fault-free. Node i then runs algorithm A, and may output a value that is outside the range of fault-free inputs if i considers a Byzantine node j's value, hence, violating validity.

7 Summary

This paper studies Byzantine consensus problems in the abstract MAC layer. We present MAC-BAC, a Byzantine approximate consensus algorithm, and MAC-RBC, a Byzantine randomized binary consensus algorithm. Both algorithms do not require the knowledge of n. To achieve so, we rely on the notion of implicit quorum. Therefore, our analysis is sufficiently different from prior work. One interesting open problem is the lower bound on the resilience of Byzantine consensus algorithms.

References

1 Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In *Principles of Distributed Systems, 8th International Conference, OPODIS 2004, Grenoble, France, December 15-17, 2004, Revised Selected Papers*, pages 229–239, 2004. doi:10.1007/11516798_17.

2 Eduardo A. Alchieri, Alysson Neves Bessani, Joni Silva Fraga, and Fabíola Greve. Byzantine consensus with unknown participants. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 22–40, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-92221-6_4.

3 Hagit Attiya and Jennifer Welch. *Distributed computing: Fundamentals, Simulations, and Advanced topics*, volume 19. John Wiley & Sons, 2004.

4 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. Association for Computing Machinery. doi:10.1145/800221.806707.

5 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. Cryptology ePrint Archive, Paper 2020/851, 2020. URL: https://eprint.iacr.org/2020/851.

6 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

7 Christian Cachin and Luca Zanolini. Asymmetric asynchronous byzantine consensus. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2021 International Workshops, DPM 2021 and CBT 2021, Darmstadt, Germany, October 8, 2021, Revised Selected Papers*, pages 192–207, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-93944-1_13.

8 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.

9 Ran Cohen, Pouyan Forghani, Juan Garay, Rutvik Patel, and Vassilis Zikas. Concurrent asynchronous byzantine agreement in expected-constant rounds, revisited. Cryptology ePrint Archive, Paper 2023/1003, 2023. URL: https://eprint.iacr.org/2023/1003.

10 Tyler Crain. A simple and efficient asynchronous randomized binary byzantine consensus algorithm, 2020. arXiv:2002.04393.

11 Danny Dolev, Cynthia Dwork, and Larry J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987. doi:10.1145/7531.7533.

12 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, 1986. doi:10.1145/5925.5931.

13 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985. doi:10.1145/3149.214121.

14 Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 246–257, 2022. doi:10.1109/ICDCS54860.2022.00032.

15 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.

16 Fabiola Greve and Sebastien Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 82–91, 2007. doi:10.1109/DSN.2007.61.

17 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.

18 Pankaj Khanchandani and Roger Wattenhofer. Brief announcement: Byzantine agreement with unknown participants and failures. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 178–180, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405740.

19 Fabian Kuhn, Nancy Lynch, and Calvin Newport. The abstract mac layer, 2011. doi:10.1007/S00446-010-0118-0.

20 Fabian Kuhn, Nancy A. Lynch, and Calvin C. Newport. The abstract MAC layer. In Idit Keidar, editor, *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, volume 5805 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2009. doi:10.1007/978-3-642-04355-0_9.

21 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.

22 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

23 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.

24 Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with t < n/3 and o(n2) 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.

25 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.

26 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.

27 Michael O. Rabin. Randomized byzantine generals. *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409, 1983. doi:10.1109/SFCS.1983.48.

28 T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distrib. Comput.*, 2(2):80–94, jun 1987. doi:10.1007/BF01667080.

29 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.

30 Lewis Tseng and Callie Sardina. Byzantine consensus in abstract mac layer, 2023. arXiv:2311.03034.

31 Lewis Tseng and Qinzi Zhang. Brief announcement: Computability and anonymous storage-efficient consensus with an abstract mac layer. In *PODC '22: ACM Symposium on Principles of Distributed Computing, Italy, 2022*. ACM, 2022. doi:10.1145/3519270.3538462.

Discrete Incremental Voting

Colin Cooper
Department of Informatics, King's College, University of London, UK

Tomasz Radzik
Department of Informatics, King's College, University of London, UK

Takeharu Shiraga
Department of Information and System Engineering, Faculty of Science and Engineering, Chuo University, Tokyo, Japan

Abstract

We consider a type of pull voting suitable for discrete numeric opinions which can be compared on a linear scale, for example, 1 ("disagree strongly"), 2 ("disagree"), ..., 5 ("agree strongly"). On observing the opinion of a random neighbour, a vertex changes its opinion incrementally towards the value of the neighbour's opinion, if different. For opinions drawn from a set $\{1, 2, \ldots, k\}$, the opinion of the vertex would change by $+1$ if the opinion of the neighbour is larger, or by -1, if it is smaller.

It is not clear how to predict the outcome of this process, but we observe that the total weight of the system, that is, the sum of the individual opinions of all vertices, is a martingale. This allows us analyse the outcome of the process on some classes of dense expanders such as complete graphs K_n and random graphs $G_{n,p}$ for suitably large p. If the average of the original opinions satisfies $i \le c \le i+1$ for some integer i, then the asymptotic probability that opinion i wins is $i+1-c$, and the probability that opinion $i+1$ wins is $c-i$. With high probability, the winning opinion cannot be other than i or $i+1$.

To contrast this, we show that for a path and opinions $0, 1, 2$ arranged initially in non-decreasing order along the path, the outcome is very different. Any of the opinions can win with constant probability, provided that each of the two extreme opinions 0 and 2 is initially supported by a constant fraction of vertices.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Random distributed processes, Pull voting

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.10

Related Version A part of this paper appeared in the 2023 ACM Symposium on Principles of Distributed Computing (PODC 2023) as a brief announcement.
Full Version: `https://arxiv.org/abs/2305.15632`

Funding *Colin Cooper*: Research supported at the University of Hamburg, by a Mercator fellowship from DFG – Project 491453517.
Takeharu Shiraga: Research supported by JSPS KAKENHI Grant Number 23K16840.

1 Introduction

Background on distributed pull voting. Distributed voting has applications in various fields of computer science including consensus and leader election in large networks [7, 13]. Initially, each vertex has some value chosen from a set S, and the aim is that the vertices reach consensus on (converge to) the same value, which should, in some sense, reflect the initial distribution of the values. Voting algorithms are usually simple, fault-tolerant, and easy to implement [13, 14].

Pull voting is a simple form of distributed voting in connected graphs. At each step, a randomly chosen vertex (asynchronous process), or each vertex (synchronous process), replaces its opinion with that of randomly chosen neighbour. The probability a particular

opinion, say opinion A, wins is $d(A)/2m$, where $d(A)$ is the sum of the degrees of the vertices initially holding opinion A, and m is the number of edges in the graph; see Hassin and Peleg [13] and Nakata *et al.* [16]. The pull voting process can be modified to consider two or more opinions at each step. The aim of this modification is twofold; to ensure the majority (or plurality) wins, and to speed up the run time of the process. Work on *best-of-k* models, where a vertex replaces its opinion with the opinion most represented in a sample of k opinions, includes [1, 4, 3, 5, 6, 8, 10, 12, 15, 18].

The general model of pull voting regards the opinions as incommensurate, and thus not comparable on a numeric scale. In contrast to this, Doerr at al. [11] consider opinions drawn from an ordered set and a process which aims to converge to the median. At each step a random vertex selects two neighbours and replaces its opinion by the median of all three values (including its own current value).

In this paper we consider another variant of pull voting, with opinions comparable on a linear scale. This variant – *discrete incremental voting* – can be seen as modelling the convergence to consensus of a group opinion, based on compromise during extended discussion. The final opinion may not be one held originally by anyone, but would reflect the compromise. If the initial (degree-weighted) average of the opinions is c, then the expectation of the final opinion (a random variable) is always equal to the initial average c. Furthermore, for some classes of expanders, w.h.p.[1] the process converges to an integer average, $\lfloor c \rfloor$ or $\lceil c \rceil$. Seen in this context, the pull voting processes above mirror the statistical measures of Mode, Median and Mean, for pull voting, median voting and discrete incremental voting, respectively.

Distributed processes for computing the exact average of the vertex values have been widely proposed and studied, but they would require calculating and storing fractional numbers and coordinated two-sided updates (both interacting vertices simultaneously update their states). The one-sided updates of pull-voting processes is an appealing simplicity.

Discrete incremental voting: An introduction. We assume the initial opinions of the vertices are chosen from among the integers $\{1, 2, ..., k\}$. As a simple example, suppose the entries reflect the views of the vertices about some issue, and range from 1 ('disagree strongly') to k ('agree strongly'). Then it seems unrealistic that a vertex would change its opinion to that of a neighbour, (as in pull voting), based only on observing what the neighbour thinks. However, people being what they are, it seems possible that they may modify their opinion slightly towards the opinion of their neighbour on observing it.

In the simplest case, suppose that a vertex v has opinion i and observes at its neighbour u opinion j. If $j > i$, then vertex v modifies its opinion to $i + 1$ (tends to agree more). Similarly, if the observed neighbour u has value $j < i$, vertex v changes its opinion to $i - 1$ (tends to disagree more). The neighbour u does not change its opinion at this interaction. That this process converges, and the value it converges to, is the topic of this paper.

We consider two related asynchronous and one synchronous variants of incremental voting. Given a connected graph $G = (V, E)$ with n vertices and m edges, let $X(t) = (X_v(t) : v \in V)$ be the vector of integer opinions at step t. The value of $X(t + 1)$ is obtained as follows.

Asynchronous vertex process: Given $X = X(t)$, pick a vertex v uniformly at random (u.a.r.) and an adjacent edge (v, w) u.a.r. The following update rule $X_v \to X'_v$ holds,

$$\left.\begin{array}{lcl} X_v < X_w & \Longrightarrow & X'_v = X_v + 1 \\ X_v = X_w & \Longrightarrow & X'_v = X_v \\ X_v > X_w & \Longrightarrow & X'_v = X_v - 1 \end{array}\right\} \tag{1}$$

[1] With high probability, which in this paper means probability $1 - o(1)$.

Asynchronous edge process[2]: Pick a random endpoint v of a random edge $e = (v, w)$ selected u.a.r. The value X_v at vertex v is updated to X'_v using the rules in (1) above.

Synchronous vertex process: Given $X = X(t)$, each vertex v picks an adjacent edge (v, w) u.a.r. and updates its value $X_v \to X'_v$ according to (1).

Discrete incremental voting: Main results. If the initial set of opinions is $\{0, 1\}$ (or $\{i, i+1\}$ for an integer i) the incremental voting (with updates (1)) is equivalent to ordinary "two-value" pull voting as studied by [13] and others: when a vertex v updates its value using the value at a neighbour w, vertex v simply takes on the value from vertex w. The "two-value" pull voting comes in the same three variants: asynchronous or synchronous vertex process, or asynchronous edge process. In a vertex process (asynchronous or synchronous), the probability that opinion 0 wins, $d(0)/2m$, is proportional to the sum $d(0)$ of the degrees of the vertices initially holding this opinion. Discrete incremental voting generalises two-value pull voting, where the simplest case differing from pull voting is when opinion values are in $\{0, 1, 2\}$. In general we assume the initial values are in the range $\{0, 1, ..., k\}$ or $\{1, ..., k\}$, where appropriate bounds on k may be required in the analysis.

In order to reach a consensus opinion, all other opinions must be eliminated. The only way to irreversibly reduce the number of opinions, is to remove one of the extreme values in the order, leading to the next stage. The process continues through such stages until one opinion remains. Returning to our original example (given at the start of the paragraph "Discrete incremental voting: An introduction", with $k = 5$), the values have the following meanings: 1 ("disagree strongly"), 2 ("disagree"), 3 ("indifferent"), 4 ("agree"), 5 ("agree strongly"). Suppose we start with initial opinions $\{1, 2, 5\}$. Then a possible evolution of the system is

$$\{1, 2, 5\} \to \{1, 2, 3, 4\} \to \{2, 4\} \to \{2, 3\} \to \{3\},$$

where the sets of opinions at the beginning of each stage are indicated, and each "$\to$" represents a sequence of one or more steps constituting one stage. The intermediate values may disappear but then they appear again (in the above example, opinion 3 disappeared in stage 2 and they appeared again in stage 3). Eventually, as extreme values disappear, we reach the final stage of voting when only two adjacent values remain. In the example above the final stage has values $\{2, 3\}$. At this point the process reverts to ordinary two-value pull voting. Suppose only values $\{i, i+1\}$ remain. Let A_j, $j \in \{i, i+1\}$, be the set of vertices with value j at the start of this final stage, and $N_j = |A_j|$, so $N_i + N_{i+1} = n$. Let $d(A) = \sum_{v \in A} d(v)$ be the total degree of set A. The probability that i wins is

$$\mathbb{P}(i \text{ wins}) = \frac{N_i}{n} \quad \text{(Edge process)}, \qquad \mathbb{P}(i \text{ wins}) = \frac{d(A_i)}{2m} \quad \text{(Vertex process)}. \tag{2}$$

In Section 2, Lemma 4, we prove that the average weight of the process is a martingale in both the asynchronous and synchronous processes. (The vertex opinion, value, and weight refer to the same quantity.) This allows us to establish Theorem 5 which gives the distribution of winning opinions in the case where the initial average c is maintained throughout the process. In this case, when only two opinions $\{i, i+1\}$ remain, we have $i \le c \le i+1$ and their winning probability is determined as in (2) above.

As shown in Section 2, Lemma 6, the expected time for one of the two extreme opinions to disappear is $O(T_2)$, where T_2 is the (worst-case) expected time to consensus for two-value pull voting on the same graph. Thus the expected time to consensus is $O(kT_2)$. See [9] and

[2] The edge process is as a vertex process but with the leading vertex v sampled with prob. $\pi_v = d(v)/2m$.

[17] for graph specific bounds on the value of T_2, e.g., for the synchronous vertex model, $T_2 = O(n^3)$ for any connected graph, $T_2 = O(n^2)$ for regular graphs, and $T_2 = O(n)$ for regular expander graphs, with bounds for asynchronous models higher by an $O(n)$ factor. However, for the complete graph K_n and some classes of expanders[3], as the number of opinions k increases, the bound $O(kT_2)$ on consensus time becomes weak. For such graphs we show that with high probability the extreme values disappear faster than the time to complete two-value pull voting. Thus, with suitable bounds on k, the expected run time can be reduced from $O(kT_2)$ to $O(T_2)$, and is directly comparable with ordinary pull voting. See e.g., Lemma 10. Ideally (for easier analysis) we would like one of the two extreme opinions to disappear completely before moving on to considering the next extreme opinion. However, to obtain good bounds, in some cases we have to move on to, say, the next smallest opinion κ, while some small number of vertices may still hold opinions smaller than κ.

As the average opinion is a martingale (details of this are in the next Section 2), in cases where the process converges rapidly to two neighbouring states, martingale concentration allows us to use Theorem 5 to predict the outcome of the process. This is fundamental for our analysis on expander graphs. For the cases we studied, $G_{n,p}$ and K_n, essentially the process converges quickly to two neighbouring states $\{i, i+1\}$. Because the time to consensus in the final stage is determined by known results, i.e., two-value pull voting, we only need to estimate the time to reach a final pair of values $\{i, i+1\}$ where w.h.p. $i \le c \le i+1$. The overall expected time to consensus is determined by the (slower) final stage of two-value pull voting; namely $O(n)$ for the synchronous, and $O(n^2)$ for the asynchronous process.

We illustrate incremental voting using three examples: the asynchronous process on $G_{n,p}$ (Theorem 2) and the synchronous process on K_n (Theorem 1), both of which work as one might expect, and an asynchronous process on the path which does not (Theorem 3).

Notation. For functions $a = a(n)$ and $b = b(n)$, $a \sim b$ denotes $a = b(1 + o(1))$, where $o(1)$ is a function of n which tends to zero as $n \to \infty$. We use ω to denote a generic quantity tending to infinity as $n \to \infty$, but suitably slowly as required in the given proof context. An event A on an n-vertex graph holds with high probability (w.h.p.), if $\mathbb{P}(A) = 1 - o(1)$.

▶ **Theorem 1.** Synchronous incremental voting on K_n.
Let the initial values of the vertices of K_n be chosen from $\{1, 2, \ldots, k\}$, where $k = o(n/(\log n)^2)$, and let $S(0) = \sum_{v \in V} X_v(0) = cn$.

- **(i)** *If $i < c < i+1$, then $\mathbb{P}(i \text{ wins}) \sim i + 1 - c$ and $\mathbb{P}(i+1 \text{ wins}) \sim c - i$. If $c = i(1 + o(1))$, then $\mathbb{P}(i \text{ wins}) \sim 1$.*
- **(ii)** *The number of opinions is reduced to at most three consecutive values in $O(k \log n)$ steps w.h.p., and the expected time for the whole process to finish is $O(n)$.*

A similar analysis for the asynchronous process, giving w.h.p. convergence to three adjacent values in $O(nk \log n)$ steps is given in the full version. The expected time for the asynchronous process to finish is $O(n^2)$.

▶ **Theorem 2.** Asynchronous incremental voting on random graphs.
Let $G \in G_{n,p}$, where $np \ge \log^{1+\varepsilon} n$ for some constant $\varepsilon > 0$. Let the initial values be in $\{1, 2, ..., k\}$ where k is a fixed positive integer, and $S(0) = \sum_{v \in V} X_v(0) = cn$ be the initial total weight.

- **(i)** *If $i < c < i+1$, then $\mathbb{P}(i \text{ wins}) \sim i + 1 - c$, and $\mathbb{P}(i+1 \text{ wins}) \sim c - i$. If $c = i(1 + o(1))$, then $\mathbb{P}(i \text{ wins}) \sim 1$.*
- **(ii)** *The expected time for the asynchronous process to finish is $O(n^2)$.*

[3] We view expansion in terms of the relative number of edges between sets S and $V \setminus S$.

The intuitive basis of Theorems 1 and 2 is to prove that the "extremal" values from $\{1, 2, ..., k\}$ disappear rapidly leaving just two values $i, i+1$ whose weighted average is c, and to which we can apply the results of two-value pull voting. For K_n this is essentially what happens, and for $G_{n,p}$ it is a reasonable approximation. We remark that the expected time to complete two-value pull voting on K_n and $G_{n,p}$ is $\Theta(n)$ in synchronous, and $\Theta(n^2)$ in asynchronous model (see [2] Chapter 14.3.3 and [9]) and the completion time of incremental voting is asymptotically of the same order.

To complement the above results, for graphs which are not expanders, we give an example on the path graph for which the final answer is quite different than in Theorems 1 and 2. Let P_n be the path with vertex set $\{1, 2, ..., n\}$, with initial values $\{0, 1, 2\}$ ordered on the path vertices in non-decreasing value: first $N_0 \geq 0$ zeroes, then $N_1 \geq 0$ ones and finally $N_2 \geq 0$ twos, where $N_0 + N_1 + N_2 = n$. We refer to such an arrangement as the ordered path.

▶ **Theorem 3.** Asynchronous incremental voting on the ordered path P_n.
If initially $N_0 = an, N_1 = (1 - (a + b))n$, *and* $N_2 = bn$, *then*
$\mathbb{P}(\textit{Opinion 0 wins}) \sim a(1 - b)$,
$\mathbb{P}(\textit{Opinion 1 wins}) \sim ab + (1 - a)(1 - b)$,
$\mathbb{P}(\textit{Opinion 2 wins}) \sim (1 - a)b$.

An example for comparison. We consider values in $\{0, 1, 2\}$. Initially 1/5 of the values are 0, none are 1, and 4/5 are 2. Thus $c = 8/5$ and $1 < c < 2$. In K_n and $G_{n,p}$,

$$\mathbb{P}(0 \text{ wins}) \sim 0, \qquad \mathbb{P}(1 \text{ wins}) \sim 2/5, \qquad \mathbb{P}(2 \text{ wins}) \sim 3/5,$$

whereas on the ordered path

$$\mathbb{P}(0 \text{ wins}) \sim 1/25, \qquad \mathbb{P}(1 \text{ wins}) \sim 8/25, \qquad \mathbb{P}(2 \text{ wins}) \sim 16/25.$$

2 Basic properties of incremental voting

Let $X(t) = (X_v(t) : v \in V)$ be the vector of integer opinions held by the vertices at step t; $X(0)$ is the vector of initial opinions. $A_i(t) = \{v \in V : X_v(t) = i\}$ is the set of vertices holding opinion $i \in \{1, ..., k\}$ at time t, and $N_i(t) = |A_i(t)|$. We may abbreviate by dropping the step index t, e.g., N_i and N_i' would refer to the number of vertices holding opinion i at the beginning and end, respectively, of the current step. Let $S(t)$ be the total weight at step $t \geq 0$: $S(t) = \sum_{v \in V} X_v(t) = \sum_j jN_j(t)$. Let $\pi_v = d(v)/2m$ where m is the number of edges of the graph, and let $Z(t) = n \sum_{v \in V} \pi_v X_v(t)$ be the degree biased weight. For regular graphs, $\pi_v = 1/n$, so $S(t) = Z(t)$. We also use notation $\|\pi\|_2 = \sqrt{\sum_i \pi^2}$ and $\|\pi\|_\infty = \max_{v \in V} \pi_v$.

A random variable $W(t), t = 0, 1, ...$ of the incremental voting process is a martingale if its expected value at the next step depends only on the current opinions $X(t)$, and it satisfies $\mathbf{E}(W(t+1) \mid X(t)) = W(t)$.

▶ **Lemma 4.** The average weight is a martingale. *The following hold for each* $t \geq 0$.
(i) Asynchronous edge process. *For arbitrary graphs,* $S(t)$ *is a martingale.*
(ii) Asynchronous vertex process. *For arbitrary graphs,* $Z(t)$ *is a martingale.*
(iii) Synchronous vertex process. *For arbitrary graphs,* $Z(t)$ *is a martingale.*

Proof.

Proof of (i). Consider step $t + 1$, take any edge (v, w) and let $\Delta_v(w)$ be the change in X_v if this edge and its endpoint v are chosen in this step. Thus $\Delta_v(w) \in \{-1, 0, +1\}$ and $\Delta_v(w) = -\Delta_w(v)$; see (1). Only one of these changes can occur at a given step in the asynchronous process. Let $e = (v, w)$ be the chosen edge, an event of probability $1/m$ in the edge process. Then,

$$\mathbf{E}(S(t+1) \mid X(t), e=(v,w) \text{ chosen}) = S(t) + \frac{1}{2}\Delta_v(w) + \frac{1}{2}\Delta_w(v) = S(t).$$

Proof of (ii). Let $A_i(t)$ be the vertices with value i at step t. For a vertex $u \in A_i$, let $s_{ij}(u)$ be the number of edges from u to A_j. Adding the edges between A_i and A_j in two ways,

$$\sum_{u \in A_i} s_{ij}(u) = \sum_{v \in A_j} s_{ji}(v). \tag{3}$$

For $1 \le i < j \le k$, let Δ_{ij} be the change in $Z(t)$ at step $t+1$ arising from an interaction between A_i and A_j (a vertex with opinion i picks up a neighbour with opinion j, or vice versa). We have $Z(t+1) = Z(t) + \sum_{i<j} \Delta_{ij}$. In the vertex process, we first pick a vertex u u.a.r and then an edge (u,v) from u u.a.r.. If $u \in A_i$ is the sampled vertex, the probability an edge from u to A_j is chosen is $s_{ij}(u)/d(u)$. As $\pi_u = d(u)/2m$, and then using (3),

$$\mathbf{E}\Delta_{ij} = \sum_{u \in A_i} \frac{1}{n}\frac{s_{ij}(u)}{d(u)}\pi_u - \sum_{v \in A_j} \frac{1}{n}\frac{s_{ji}(v)}{d(v)}\pi_v = \frac{1}{2nm}\left(\sum_{u \in A_i} s_{ij}(u) - \sum_{v \in A_j} s_{ji}(v)\right) = 0. \tag{4}$$

Proof of (iii). In the synchronous vertex process, with the notation as above in *(ii)*, the expected value of Δ_{ij} is as in (4) but without the $1/n$ factors (since each vertex selects a neighbour and updates its value). ◀

As the process is randomized, the final value is a random variable with distribution $D(i)$ on the initial values $\{1, ..., k\}$. The following theorem helps us to characterize this distribution in certain cases. If only two neighbouring opinions $i, i+1$ remain at some step t, the process is equivalent to two-value pull voting, and we say the voting is at the final stage.

▶ **Theorem 5.** DISTRIBUTION OF WINNING VALUE. *Let $W(t)$ stand for $S(t)$, if we refer to the edge model, or for $Z(t)$, if we refer to the vertex model. Let $W(0) = cn$ be the total initial weight, where c is the initial average opinion.*

- **(i)** *For any graph, the expected average opinion at any step is always the initial average: $\mathbf{E}[W(t)/n] = W(0)/n = c$. $W(t)$ converges to a time invariant random variable.*
- **(ii)** *If at the start of the final stage only two opinions i and $i+1$ remain and the total weight W is $c'n$, then for any connected graph, the winning opinion is i with probability $p = i + 1 - c'$, or $i+1$ with probability $q = c' - i$.*
- **(iii)** *Suppose the final stage is reached in T steps, where $T = o(1/\|\pi\|_2^2)$ for the synchronous vertex process (which reduces to $T = o(n)$ for regular graphs), $T = o(n^2)$ for the asynchronous edge process, and $T = o(1/\|\pi\|_\infty^2)$ for the asynchronous vertex process. Then w.h.p. $|W(T) - W(0)| = o(n)$ and the results of part (ii) hold with $c' \sim c$. That is, for i such that $i \le c < i+1$, the winning opinion is i with probability $p \sim i + 1 - c$, or $i+1$ with probability $q \sim c - i$.*

Proof. **(i)** The first part follows from $\mathbf{E}W(t) = W(0)$ (Lemma 4). $\mathbf{E}W^2(t) \le k^2$ and the limit random variable, for $t \to \infty$, exists by the martingale convergence theorem.
(ii) Using (2), we have $ipn + (i+1)qn = W$, implying that $p = i + 1 - c'$ and $q = c' - i$.
(iii) In the synchronous vertex process, using Lemmas 12 and 13, we have $|W(T) - W(0)| = o(n)$ w.h.p., provided $T = o(1/\|\pi\|_2^2)$, which reduces to $T = o(n)$ for regular graphs. In the asynchronous edge process, $|S(t+1) - S(t)| \le 1$, and in the asynchronous vertex process,

$|Z(t+1) - Z(t)| \leq n \max_{v \in V} \pi_v = n\|\pi\|_\infty$, so using the Azuma-Hoeffding inequality for martingale concentration (Lemma 18), we obtain $|W(T) - W(0)| = o(n)$ w.h.p., provided $T = o(n^2)$, respectively $T = o(1/\|\pi\|_\infty^2)$. ◀

In the light of Theorem 5, there are two main ways of analysing the problem. On the (regular) expander graphs we consider the final stage of pull voting with two values i and $i+1$, takes $T = O(n)$ expected time (synchronous process) or $T = O(n^2)$ expected time (asynchronous process); see e.g. [2, 9]. If the time t for the other opinions to disappear, leaving only two neighbouring values, can be shown to be $t = o(T)$ then the total weight $S(t)$ is concentrated around $S(0) = cn$. In this case we can use Theorem 5 to give an asymptotic result on the distribution D. There are however challenges in the analysis, e.g., dealing with the fact that a small number of opinions other than i or $i+1$ may persist in the system for a longer time. This is the topic of Sec. 3 and 5. On graphs which are not expanders we cannot expect $S(t)$ (or $Z(t)$) to be concentrated, but we can try to explicitly obtain the distribution D of the limiting random variable, at least for some special cases. This is the topic of Sec. 4.

In all cases, the winning value of an incremental voting process will be given by a probability distribution D on $\{1, ..., k\}$. As the total weight $W(t)$ is a martingale, the distribution D must have expected value $c = W(0)/n$. Indeed, for the vertex process and $Z(t)$ (and similarly for the edge process and $S(t)$),

$$c = \frac{Z(0)}{n} = \lim_{t\to\infty} \frac{\mathbf{E}Z(t)}{n} = \sum_v \pi_v \lim_{t\to\infty} \mathbf{E}X_v(t) = \sum_v \pi_v \mathbf{E}D = \mathbf{E}D. \tag{5}$$

▶ **Lemma 6.** Completion time, a general bound. *For any connected graph and any of the three types of incremental voting (asynchronous or synchronous vertex process, or asynchronous edge process), the worst-case expected time to eliminate one of the two extreme opinions (over all initial configurations) is at most the worst-case expected completion time of the corresponding (asynchronous, synchronous vertex, or asynchronous edge, respectively) standard 2-opinion voting process.*

Proof. Let $A_i = \{v \in V : X_v = i\}$. We consider our process $(A_1(t), A_1(t), \ldots, A_k(t))_{t\geq 0}$ and the standard 2-opinion voting $B(t)_{t\geq 0}$, where $B(t)$ and $V \setminus B(t)$ are the supports of the two opinions at time t.

We set $B(0) = A_\ell(0)$, where ℓ is the minimum opinion: $\ell = \min\{\kappa : A_\kappa(0) \neq \emptyset\}$, and couple processes A and B, running them on the same random selection of vertices in each step. The two opinions in process B are opinions ℓ and non-ℓ, that is, for process B, each opinion other than opinion ℓ is viewed as the same opinion non-ℓ. While initially the vertices with opinion ℓ in process B are exactly the vertices with this opinion in process A, this does not need to be the case later during the computation. If in the first step a vertex v with A-opinion (its opinion in process A) equal to $q \geq \ell + 2$ picks up a neighbour with A-opinion ℓ, then it updates its A-opinion to $q - 1 \geq \ell + 1$, but its B-opinion becomes ℓ.

Throughout the computation, however, the following relations for the two extreme opinions ℓ and $r = \max\{\kappa : A_\kappa(0) \neq \emptyset\}$ hold by induction,

$$A_\ell(t) \subseteq B(t), \quad A_r(t) \subseteq V \setminus B(t) \tag{6}$$

If at step t the vertex v changes its A-opinion to ℓ, (and consequently $v \in A_\ell(t+1)$), this happens because $v \in A_{\ell+1}(t)$ and picks a neighbour $w \in A_\ell(t)$. Then we must also have $v \in B(t+1)$, because by induction $w \in B(t)$, so its B-opinion is ℓ. Similarly, if $v \in A_{r-1}(t)$ changes its A-opinion to r, this happens because v picks a neighbour $w \in A_r(t)$; By induction $w \in V \setminus B(t)$, and so $v \in V \setminus B(t+1)$.

Let T be the step when the two-voting process B completes, that is, the first step when $B(T)$ is either empty or the whole set V. In the former case, $A_\ell(T) = \emptyset$ and in the latter $A_r(T) = \emptyset$, from (6), so by step T, either opinion ℓ or r must have been eliminated. ◀

▶ **Corollary 7.** *The expected completion time of the discrete incremental voting is $O(k \cdot \mathcal{T}_{2-vote})$, where $\mathcal{T}_{2-vote}$ is the worst-case expected completion time of the 2-opinion voting.*

3 Analysis of asynchronous process for $G_{n,p}$: proof of Theorem 2

In this section we analyse the asynchronous incremental voting on random graphs $G_{n,p}$ above the connectivity threshold, which we view as examples of expanders. Much of the analysis is general, so the results should hold equally for other classes of graphs with expansion properties similar to those in Lemma 8. To indicate why this should be the case, we use the example of the asynchronous edge process on a d-regular graph with opinions in $\{0, 1, 2\}$. The expected change in the two 'extremal' values $0, 2$ at any step is given by

$$\mathbf{E}(N_0' + N_2') = N_0 + N_2 - \frac{2}{dn} M_{0,2}, \tag{7}$$

where $M_{0,2}$ is the number of edges between vertices holding opinion 0 and those holding opinion 2. This is because when an edge in $M_{0,2}$ is selected, then $N_0 + N_2$ is reduced by 1, and when an edge in $M_{0,1} \cup M_{1,2}$ is selected, then the expected change of $N_0 + N_2$ is zero.

For a d-regular connected graph G and any two vertex sets S and T in G,

$$\left| e(S,T) - \frac{d|S||T|}{n} \right| \le \lambda \sqrt{|S|(1-|S|/n)|T|(1-|T|/n)}, \tag{8}$$

where $e(S,T)$ is the number of edges between S and T and λ is the absolute value of the second eigenvalue of the adjacency matrix of G. Assume that $\lambda = \varepsilon d$ for some constant $\varepsilon < 1$ (the expander assumption) and take $|S| = N_0$ and $|T| = N_2$. If N_0 and N_2 are large, then so is $M_{0,2} = e(A_0, A_2)$, since from (8), $M_{0,2}$ is close to dN_0N_2/n. Thus from (7), $N_0 + N_2$ quickly decreases. Although this type of argument may not allow us to completely eliminate one of the extremal values 0 and 2 (as $M_{0,2}$ becomes too small, eventually 0, while both A_0 and A_2 are still non-empty), we can use it to make one of N_0 and N_2 small relative to N_1.

Returning to graphs $G_{n,p}$, we assume all opinions are in $\{1, 2, \ldots, k\}$, for an arbitrary but fixed integer k (constant, while n grows to infinity). The entire point of the proof in this section is to ensure that within $T = o(n^2)$ steps, all but $o(n)$ vertices have two adjacent opinions in $\{i, i+1\}$. This will allow us to apply Theorem 5 (ii)-(iii), with $c' \sim c$.

In the edge process (and in the vertex process in regular graphs) the expected change in the number of vertices with any given value can be characterised as follows. Let $M_{i,j} = M_{i,j}(t)$ be the number of edges between sets A_i and A_j at step t. Letting $N_i = N_i(t)$ and $N_i' = N_i(t+1)$,

$$\mathbf{E}N_i' = N_i + \frac{1}{2m} \left(\sum_{j \ge i+1} M_{i-1,j} + \sum_{j \le i-1} M_{i+1,j} - \sum_{j \ne i-1,i,i+1} M_{i,j} \right). \tag{9}$$

If $k > 2$, then the number of vertices with an extreme value 1 or k exhibits downward drift (the first two sums in (9) are equal to 0 for i equal to 1 or k), provided there are edges between A_1 and $\bigcup_{j \ge 3} A_j$, or between A_k and $\bigcup_{j \le k-2} A_j$. Thus if there is enough connectivity (expansion) in the graph, then the support of one of the extremal values reduces relatively quickly to $o(n)$. If this was, say, opinion 1, then, still relatively quickly, the support of the next extremal opinion, either 2 or k, reduces to $o(n)$; and so on, until the support of all opinions other than some two consecutive opinions i and $i+1$ is reduced to $o(n)$. The analysis of the completion of the process from such a state will requires another approach.

In what follows, ω and ω' denote functions tending to infinity with n, with $\omega' = o(\omega)$. In general the exact values are not important but the growth to infinity has to be sufficiently slow to satisfy the bounds arising in the analysis. In the final part of the analysis, we will choose $\omega = \log n$, and $\omega' = \log\log n$.

Required properties of $G_{n,p}$. The following are the expansion properties of $G_{n,p}$ needed for our proofs. The lower bound on np ensures that w.h.p. the graph is connected. To maintain continuity of discussion the proof of the following lemma is given in Appendix A.

▶ **Lemma 8.** *Let $G \in G_{n,p}$, where $np \geq \log^{1+\varepsilon} n$ for some constant $\varepsilon > 0$. The following properties hold w.h.p..*

P1. (Almost regular graphs) *G is connected and all vertices v have degree $d(v) = np + O(\sqrt{np\log n})$ and stationary distribution $\pi_v = \frac{1}{n} + O\left(\frac{1}{n\log^{\varepsilon/2} n}\right)$.*

P2. (Large number of edges between large subsets of vertices) *Let $\delta \geq 5/\sqrt{np}$. For any pair of disjoint vertex sets A, B, with $|A| \geq \delta n$, $|B| \geq \delta n$, the number of edges X_{AB} between A and B satisfies $\mu/2 \leq X_{AB} \leq 3\mu/2$, where $\mu = |A||B|p$, the expected number of edges between the sets A and B in $G_{n,p}$.*

P3. (Not too many edges within small subsets of vertices)

- **(i)** *For $\omega \geq e$, $\omega\log\omega \leq np$, no vertex set S of size $s = n/\omega$ induces more than $X_S = e^2 s^2 p$ edges.*
- **(ii)** *Let $d = np$. No set S, $|S| \leq n/\omega$ induces more than $X_S = s\sqrt{4d\log(ne/s)}$ edges.*
- **(iii)** *Provided $\omega = O(\log n)$ and $np = d \geq \log^{1+\theta} n$, the ratio $X_S/X_{S,V-S}$ is $O(1/\omega)$, the value achieved in P3.(i) above.*

Outline to the analysis of the process. Our analysis of $G_{n,p}$ is for the edge process, but as w.h.p. vertex degrees are concentrated for the range of p we consider, the vertex process and edge process are asymptotically equivalent. Indeed, by property P1 of Lemma 8, the degree weighted total $Z(t)$ and the unweighted total $S(t)$ satisfy $|Z(t) - S(t)| \leq c/\log^{\varepsilon/2} n$, and it suffices to analyse the convergence of $S(t)$.

Ideally we would like to keep completely removing the values $\{1, ..., k\}$ one by one in some order, as in the proof of Theorem 1. As can be seen from (9), the drift on the extremal values $1, k$ is negative or zero.

$$\mathbf{E}N_1' + \mathbf{E}N_k' = N_1 + N_k - \frac{1}{2m}\left(\sum_{j\geq 3} M_{1,j} + \sum_{j\leq k-2} M_{k,j}\right). \tag{10}$$

Thus at least one extremal value 1 or k should disappear, allowing us then to repeat the analysis with e.g., values $\{2, ..., k\}$. However, the time taken for such an approach is $\Omega(n^2)$, which is too long for the total weight $S(t)$ to remain concentrated around $S(0)$. Therefore, in our analysis, we settle for making one of the extremal values sufficiently small, which can be done in $o(n^2)$ steps and then repeat the analysis for the remaining large values. Finally one value dominates, and w.h.p. all other values disappear at some subsequent step. It remains to be proved below that such an approach can be made to work.

The analysis proceeds in three phases, which in outline are as follows.

I. One by one, the extremal values are made small. By the beginning of iteration r, $1 \leq r \leq k-2$, the support for $r-1 = i-1+(k-j)$ extremal values $\{1, 2, \ldots, i-1\} \cup \{j+1, j+3, \ldots, k\}$ has been made small, but $N_i > \delta_r n$ and $N_j > \delta_r n$. During iteration r, the next extremal value, either i or j, is made small. As we progress through the iterations, our analysis loses accuracy, so δ_r increases with r (but remains $o(1)$).

II. For two adjacent values i and $i+1$, $N_i, N_{i+1} > n/\omega$ and $N_i + N_{i+1} = n(1-o(1))$.
III. There is a unique value i with $N_i = n(1-o(1))$.

Arriving at Phase II, the process corresponds (in general principle) to ordinary pull voting with two values. If at the completion of Phase I $\min\{N_i, N_{i+1}\} < n/k\omega$, we skip Phase II. Phase III is a clean up phase, removing any remaining small sets. At the end of Phase III, $N_i = n$, and the analysis is completed.

Phase I. Making small the first extremal value (either N_1 or N_k).

▶ **Lemma 9.** *Let $\delta = \max\left(1/n^{1/4}, 5/\sqrt{np}\right)$. Let T_1 be the number of process steps to reduce one of A_1 or A_k to size at most δn and let α be the extreme opinion (1 or k) with the size of support at most δn at step T_1. Then the following hold w.h.p.*

(i) $\mathbf{E}T_1 = O(n^{3/2})$.
(ii) $|S(T_1) - S(0)| = o(n)$.
(iii) $N_\alpha(t) \le \omega\delta n$ *at all steps* $t > T_1$ *(for some* $\omega \to \infty$*).*

Proof. Let $A = A_1$, $B = A_k$, $|A| = N_1 > \delta n$ and $|B| = N_k > \delta n$. We proceed in stages indexed by *decreasing* ℓ. At the beginning of the current stage, we assume w.o.l.g. that $N_1 \le N_k$ and integer $\ell \ge 1$ is such that $\ell\delta n \le |A| < (\ell+1)\delta n$ (that is, $\ell = \lfloor |A|/(\delta n) \rfloor$). This stage continues until N_1 or N_k drops below $\ell\delta n$, when the next stage $\ell - 1$ starts. At the end of the final stage, for $\ell = 1$, one of N_1 or N_k is less than δn.

We estimate the expected time of one stage ℓ by first estimating the number of *active* steps in this stage, defined as the steps when N_1 changes, either by -1 or $+1$. We estimate the number of active steps by comparing the random variable N_1 with the biased random walk on the integer line. We then factor in the expected number of process steps between two consecutive active steps.

We view the random selection in the current step as selection of a uniformly random *oriented* edge (v, u), where v is the vertex which updates its value. The value of N_1 decreases, resp. increases, by 1, if and only if, the selected oriented edge belongs to $(A, \overline{A})$, resp. (A_2, A). Thus in the current *active* step N_1, the ratio of the probability q that N_1 decreases by 1 to the probability $r = 1 - q$ that N_1 increases by 1 is equal to

$$\frac{q}{r} = \frac{|(A,\overline{A})|}{|(A_2,A)|} \ge \frac{|(A,\overline{A})|}{|(\overline{A},A)| - |(B,A)|} \ge 1 + \frac{|(B,A)|}{|(A,\overline{A})|} \ge 1 + \frac{N_1\, N_k\, p/2}{N_1\, n\, 3p/2} \ge 1 + \frac{\ell\delta}{3} = \frac{1+\varepsilon}{1-\varepsilon}, \quad (11)$$

where $\varepsilon \ge \ell\delta/4$. We have used Lemma 8.P2 to bound the number of edges between sets A and B and sets A and $\overline{A}$.

For a biased random walk on the integer line $\{0, 1, ..., L\}$, where in each step the probabilities of moving left (towards 0) or right are equal to $q > 1/2$ and $r = 1 - q$, respectively, the probability q_z of ruin (absorption at zero) and the expected number d_z of steps to ruin when starting from position z are equal to

$$q_z = \frac{\rho^L - \rho^z}{\rho^L - 1} = 1 - \frac{\rho^z - 1}{\rho^L - 1}, \qquad d_z = \frac{z}{q-r}\left(1 - \frac{L}{z}\cdot\frac{\rho^z - 1}{\rho^L - 1}\right). \quad (12)$$

where $\rho = q/r > 1$. We view N_1 as a random walk on the integer line $\{\ell\delta n, \ldots, (\ell+2)\delta n\}$, starting at $\ell\delta n + z$, for $z = z(\ell) < \delta n$. At an active step, the probabilities of moving left or right are at least $q = \frac{1}{2}(1+\varepsilon)$ and at most $r = \frac{1}{2}(1-\varepsilon)$, respectively. Then the probability $q'_{z(\ell)}$ that N_1 reaches $\ell\delta n$ before it reaches $(\ell+2)\delta n$ or N_k reaches $\ell\delta n$ is such that, from (12),

$$1 - q'_{z(\ell)} \le \frac{\rho^{\delta n} - 1}{\rho^{2\delta n} - 1} = \frac{1}{\rho^{\delta n} + 1} \le \frac{1}{(1+\ell\delta/3)^{\delta n}} \le e^{-\ell\delta^2 n/4} \le e^{-\sqrt{n}/4}, \quad (13)$$

where the last inequality follows from $\delta \geq 1/n^{1/4}$. The expected duration $d'_{z(\ell)}$ of this stage, in terms of the number of active steps, is at most, from (12),

$$d'_{z(\ell)} \leq \frac{z}{q-r} \leq \frac{\delta n}{\varepsilon} = \frac{4n}{\ell}. \tag{14}$$

By Lemma 8.P2, at step t of the current stage ℓ,

$$\mathbb{P}(t \text{ is an active step}) \geq \frac{|(A, \overline{A})|}{2m} \geq \frac{(\ell\delta n)(n/2)p/2}{(1+o(1))n^2 p} \geq \frac{\ell\delta}{5} \equiv P_\ell. \tag{15}$$

Thus for the first process step T_1 at which $N_1 < \delta n$ or $N_k < \delta n$, we have, from (14) and (15),

$$\mathbf{E}(T_1) = \sum_\ell \frac{1}{P_\ell} d'_{z(\ell)} \leq \sum_{\ell \geq 1} \frac{5}{\ell\delta} \cdot \frac{4n}{\ell} = O\left(\frac{n}{\delta}\right) = O(n^{5/4}).$$

As $S(t)$ is a martingale, $\mathbf{E}S(t) = S(0)$. Apply the Azuma martingale inequality (Lemma 18) to the sequence of oriented edges $(e_1, ..., e_t)$ inspected at steps $1, \ldots, t$. At each step, S changes by at most 1. At step $T_1^* = \omega \mathbf{E} T_1$, with $h = \sqrt{3T_1^*(\omega + \log T_1^*)} = o(n)$,

$$\begin{aligned}\mathbb{P}((|S(T_1) - S(0)| \geq h) &\leq \mathbb{P}(\exists T < T_1^* : |S(T) - S(0)| \geq h) + \mathbb{P}(T_1 > T_1^*) \\ &\leq T_1^* e^{-h^2/(3T_1^*)} + o(1) = e^{-\omega} + o(1) = o(1).\end{aligned}$$

Thus w.h.p. $|S(T_1) - S(0)| = o(n)$, as required.

For part (iii) of the lemma, beyond step T_1, we bound N_α (where α is opinion 1 or k and $N_\alpha \leq \delta n$ at step T_1) with the progress of *unbiased* random walk on $\{0, 1, \ldots, L = \omega\delta n\}$ starting at $z \leq \omega\delta n$, which with probability $1 - 1/\omega$ is absorbed at zero before reaching L. ◀

Phase I. Making the next extremal value small. Having completed the first iteration, we continue inductively for general iteration g, $1 < g \leq k-2$. By the beginning of this iteration, the support for $g-1$ extremal values $\{1, 2, \ldots, i-1\} \cup \{j+1, j+2, \ldots, k\}$ has been made small, but $N_i > \delta_g n$ and $N_j > \delta_g n$. Note that $j \geq i+2$. During iteration g, the next extremal value is made small, that is, N_i or N_j is reduced to at most $\delta_g n$.

Let $\delta_1 = \delta$, and in general, for $1 < g \leq k-2$, $\delta_g = \omega^{2(g-1)}\delta$, where $\omega \to \infty$ but sufficiently slowly so that $\omega^3 \delta_{k-2} = o(1)$. The argument is similar as in Lemma 9 for the first iteration, replacing N_1, N_k and δ with N_i, N_j and δ_g, and assuming by induction that the support for the reduced $g-1$ values is, and w.h.p. will remain, at most $\omega\delta_1 n + \omega\delta_2 n + \cdots \omega\delta_{g-1} n < 2\delta_g/\omega$. As in (11), in an active step of stage ℓ of this iteration, the ratio of the probability q that N_i decreases by 1 to the probability $r = 1 - q$ that it increases by 1 is equal to

$$\frac{q}{r} = \frac{|(A_i, \overline{A_i})|}{|(A_{i+1}, A_i)| + |(A_{i-1}, A_i)|} \geq \frac{|(A_i, \overline{A_i})|}{|(\overline{A_i}, A_i)| - |(A_j, A_i)|} \geq 1 + \frac{\ell\delta_g}{3} = \frac{1+\varepsilon}{1-\varepsilon},$$

where $\varepsilon \geq \ell\delta_g/4$. Proceeding as in the proof of Lemma 9, we conclude that w.h.p.: (i) the expected time $\mathbf{E}(T_g - T_{g-1})$ of iteration g is $O(n/\delta_g) = O(n^{5/4})$; (ii) at step T_g when this iteration ends, $|S(T_g) - S(0)| = o(n)$; (iii) $N_\alpha(T_g) \leq \delta_g n$ and $N_\alpha(t) \leq \omega\delta_g n$ at all steps $t > T_g$, where α is the reduced opinion i or j. (Recall that we assume that k is constant.)

For the final point (iii), the argument is more subtle for the general iteration than it was for the first iteration. Assuming that $\alpha = i$, N_i may have a tendency to increase (a positive drift) after step T_g, if $M_{i-1,i+1}$ happens to become larger than the number of edges adjacent to A_i (see (9)). To deal with this, while there are opinions smaller than i we only care that if N_i increases to $\omega\delta_g n/2$, then the probability that it further increases to $\omega\delta_g n$ before going back below $\delta_g n$ is exponentially small (as in (13)). When i becomes the smallest surviving opinion, than we start comparing the changing N_i with the unbiased random walk on $\{0, 1, \ldots, L = \omega^2\delta_g n\}$ starting from $z \leq \omega\delta_g n$ (similalry as in the first iteration).

Phase II analysis. When Phase I ends at a step t_I, then for some i, the total support for $k-2$ values $\{1,2,\ldots,i-1\}\cup\{i+2,i+3,\ldots,k\}$ is at most $2\omega\delta_{k-2}n \le n/\omega^2$. Thus Phase II starts with $N_i(t_I)+N_{i+1}(t_I)=n(1-o(1))$. We also have $|S(t_I)-S(0)|=o(n)$ (since $S(t)$ is a martingale and $t_I=O(n^{3/2})$), so we can take $i=\lfloor c\rfloor$, where $c=S(0)/n$, and have $N_i=(i+1-c)n+o(n)$, $N_{i+1}=(c-i)n+o(n)$. During Phase II, we compare N_i with unbiased random walk on $\{0,\ldots n\}$ which is driven by selecting edges in $M_{i,i+1}$. Each such edge increases or decreases N_i by 1 with equal probability. We continue Phase II only while $\min(N_i,N_{i+1})\ge\omega^2\delta_{k-2}n$ to ensure that the opinions other than i and $i+1$ remain small and distort the unbiased random walk of N_i only in a negligible way.

Phase II ends within the expected $O(n^2)$ steps, and w.h.p. $N_{i'}\ge n-n/\omega$, and $\sum_{j\neq i'}N_j\le n/\omega$, where $i'=i$ with probability $i+1-c+o(1)$, or $i+1$ with probability $c-i+o(1)$.

Phase III analysis. At the start of Phase III, there is one opinion i for which $N_i=n(1-o(1))$. Let $A=A_i$, and $B=\cup_{j\neq i}A_j$. We show that w.h.p. opinion i wins.

The process during the final Phase III resembles a "balls in boxes" system in which a vertex with value j is a box with j balls. When a vertex is selected, then it may get one ball added or removed, depending on the number of balls in the selected neighbour. When all vertices have the same value (the same number of balls), the process ends. If an edge with values $(i,j), j\neq i$ is chosen, then we call this a Type 1 event. Choosing an edge with values $(j,j'), j\neq i\neq j'$ is a Type 2 event. We ignore (i,i) and (j,j) events.

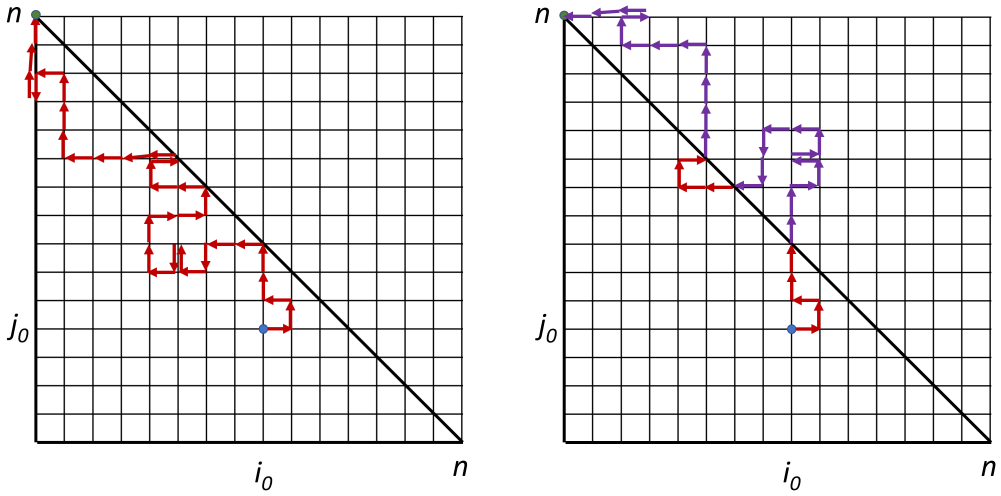
We compare this process with an unbiased walk on integers $\{0,1,...,L\}$ starting from $z=\hat{S}(0)=\sum_{j\neq i}|j-i|N_j$, the weight of set B at the start of Phase III w.r.t. value i, representing the distance between the starting configuration and the target configuration when all vertices have value i. Thus initially $z\le k|B|\le n/\omega$. Each Type 1 event changes z by $+1$ or -1 with equal probability (once an edge is selected, one of its end vertices is chosen for an update with equal probability). If only Type 1 events occurred, then z would be an unbiased random walk on $\{0,1,...,L\}$, and we put $L=\omega z$. (The process does not stop before $\hat{S}(t)$ reaches 0 or L.) Value i wins when the walk is absorbed at zero. The probability of this is $1-z/L=1-1/\omega$ and the expected duration is $z(L-z)$.

For steps $t=0,1,...,$ as Phase III proceeds, the value $\hat{S}(t)$ will change due to both Type 1 and Type 2 events. The change due to Type 1 events is directly included in the random walk given above, and with probability $1-\omega' z/n$ the walk will not increase above $\omega'\hat{S}(0)\le\omega' n/\omega$, where $\omega'\to\infty$ but $\omega'=o(\omega)$.

Type 2 events are not represented directly by the random walk. Each Type 2 occurrence can change $\hat{S}$ by $+1$ or -1. A Type 2 event on an edge (j,j'), where $j<j'$, increases or decreases the number of balls in the system by one with equal probability. After T events of Type 2, the additional change in $\hat{S}$ due to this is $(+1)X+(-1)(T-X)$ where $X\sim Bin(T,1/2)$. Thus w.h.p. X will not exceed $O(\sqrt{T\log T})$.

Only Type 1 moves can increase the size of B, whereas Type 2 moves may decrease it, if $j=i-1$ or $j'=i+1$. W.h.p. the maximum size of B due to the Type 1 walk is at most $s=\omega' n/\omega$. By Lemma 8, property P2, w.h.p. no set of size s induces more than $O(s^2p)$ edges, whereas by property P1, there are at least $nsp/3$ edges between A_i and B. Thus the probability of a Type 2 event is at most $O(s/n)$, and the number of Type 2 events in the duration of the Type 1 random walk is w.h.p. of order at most $T=zL\frac{s}{n}\omega'=O((n\omega'/\omega)^2)$. W.h.p. the maximum increase of $\hat{S}$ due to Type 2 events is $O(\sqrt{T\log T})=O(n\omega'(\log n)^{1/2}/\omega)$. Taking $\omega=\log n$ and $\omega'=\log\log n$ (so $\omega\ge(\omega')^2(\log n)^{1/2}$), we can increase $\hat{S}(0)$ to $z'=n/\omega'$ and conclude that with probability $1-1/\omega'$, at the end of Phase III $N_i=n$, as required.

It can be shown, by counting in the ignored events (i,i) and (j,j) and considering stages of halving the value z, that Phase III ends within the expected $O(n^2)$ steps.



(a) Random walk on the triangle grid. **(b)** Random walk on the square grid.

Figure 1 The random walk on the square grid (right diagram) and the corresponding walk on the triangle grid (left diagram). The random walk of the triangle grid defines the evolution of the incremental voting on the ordered path. In this example, the averaging process starts with i_0 vertices with opinion 0 and j_0 vertices with opinion 2, and stabilises with all vertices having opinion 2.

4 Asynchronous incremental voting on the line: proof of Theorem 3

To indicate that Theorems 5 and 2 do not hold for general graphs, we consider the following specific example of an *ordered path*. The graph is a path with n vertices $\{1, 2, \ldots, n\}$. There are three opinions $0, 1, 2$ and initially they are ordered along the path: vertices $\{1, \ldots, i_0\}$ have opinion 0, vertices $\{i_0 + 1, \ldots, n - j_0\}$ have opinion 1, and vertices $\{n - j_0 + 1, \ldots, n\}$ have opinion 2. Thus, the $0 \le i_0 \le n$ vertices in the initial segment of the path have opinion 0, the $0 \le j_0 \le n - i_0$ vertices in the final segment of the path have opinion 2, and the remaining $n - (i_0 + j_0)$ vertices in the middle of the path have opinion 1.

We show that the probability that opinion 0 wins is equal to $a(1 - b)$, where $a = i_0/n$ and $b = j_0/n$. By symmetry, the probability that opinion 2 wins is equal to $(1 - a)b$, leaving the probability of $ab + (1 - a)(1 - b)$ for opinion 1 to win.

The non-decreasing order of the opinions along the path is an invariant of the process, so each intermediate configuration is characterised by the number $i = N_0$ of vertices at the beginning of the path with opinion 0, and the number $j = N_2$ of vertices at the end of the path with opinion 2, where $0 \le i \le n$, $0 \le j \le n - i$. The process ends when opinion 0 wins (i becomes n), or opinion 2 wins (j becomes n), or opinion 1 wins (both i and j become 0).

The process of changing from one configuration to the next one is a random walk on the integral points of the triangle $i \ge 0, j \ge 0, i + j \le n$; see Figure 1a. Considering only the steps when the configuration changes, a configuration (i, j) which is strictly inside this triangle (that is, $i > 0, j > 0, i + j < n$) changes to any of the four configurations $(i + 1, j)$, $(i - 1, j)$, $(i, j + 1)$ and $(i, j - 1)$ with equal probability of 1/4. Indeed, configuration (i, j) changes when either edge $(i, i + 1)$ (with opinions 0 and 1) or edge $(n - j, n - j + 1)$ (with opinions 1 and 2) is selected (equal probability). If edge $(i, i + 1)$ is selected, then the configuration changes to $(i + 1, j)$ or $(i - 1, j)$, depending which vertex i or $i + 1$ updates its opinion. With equal probability, either vertex $i + 1$ decreases its opinion from 1 to 0, or vertex i increases its opinion from 0 to 1. Analogously when edge $(n - j, n - j + 1)$ is selected.

From a configuration $(0, j)$, where $0 < j < n$, we have equally probable transitions to configuration $(0, j + 1)$ or $(0, j - 1)$. Analogously, a configuration $(i, 0)$, where $0 < i < n$, transitions to $(i + 1, 0)$ or $(i - 1, 0)$ with equal probability.

Finally, consider the *diagonal* configurations lying on the side of the triangle formed by the line segment from $(0, n)$ to $(n, 0)$. For a non-final configuration (i, j): $i + j = n$, $i > 0$, $j > 0$, the vertex i has opinion 0 and vertex $i + 1$ has opinion 2. The configuration changes when the unique edge $(i, i + 1)$ is selected. In this case the configuration transitions to $(i - 1, j)$, when vertex i increases its opinion from 0 to 1, or to $(i, j - 1)$, when vertex $i + 1$ decreases its opinion from 2 to 1 (equal probability for either of these two transitions).

For convenience, we view this random walk W on the triangle as a random walk W' on the full square $0 \le i \le n, 0 \le j \le n$, unifying the pairs of states (i, j) and $(n - j, n - i)$, these being identical on the diagonal of the triangle. See Figure 1, where the right diagram gives an example of the random walk W' on the square grid, and the left diagram shows the corresponding walk on the triangle. The transition probabilities for walk W' are the same as for W for all non-diagonal states (i, j). For such a state, if it is not on the boundary of the square, then one of the coordinates increases or decreases by 1, with all four possibilities equally probable. For a state on the boundary of the square, the configuration changes, with equal probability, to one of the two neighbouring boundary states.

For a diagonal non-final state (i, j), the random walk W' moves also to any of the four neighbouring states $(i + 1, j)$, $(i - 1, j)$, $(i, j + 1)$ or $(i, j - 1)$, with equal probability. In this case, the transition of W' with probability $1/2$ to either $(i - 1, j)$ or $(i, j + 1)$ corresponds to random walk W moving with probability $1/2$ from configuration (i, j) to configuration $(i-1, j)$. Thus the pair of states $(i-1, j)$ and $(i, j+1)$ in W' correspond to the configuration $(i - 1, j)$ in W. The below diagonal state (i, j) and above diagonal state $(n - j, n - i)$ in the square, both correspond to state (i, j) in the triangle.

Thus our incremental voting on the path corresponds to the random walk W' on the square grid. Inside the square the walk transitions with equal probability from one state to any of the four neighbouring states. A transition on one coordinate is completely independent of the value of the other coordinate. When the walk hits a side of the square, this corresponds to one of the two extreme values 0 or 2 being eliminated. The walk then remains within this side of the square, moving independently to one of the two neighbouring boundary states. The final absorbing states are the four corners of the square. State $(n, 0)$ corresponds to opinion 0 winning, state $(0, n)$ corresponds to opinion 2 winning, and states $(0, 0)$ and (n, n) (corresponding to state $(0, 0)$ in the triangle) correspond to opinion 1 winning.

What is the probability that the random walk W' terminates in the state $(n, 0)$, meaning the win for opinion 0? We generate the two-dimensional random walk W' from two independent one-dimensional walks, one walk for each of the two coordinates, both walks with the range $\{0, 1, \ldots, n\}$. To move walk W', we take, with equal probability, the next step from one of the two one-dimensional walks. Walk W' ends in the state $(n, 0)$ if, and only if, the one-dimensional walk for the coordinate i ends in state n and the one-dimensional walk for the coordinate j ends in state 0. Indeed, for the "if" part, if the one dimensional random walks for coordinates i and j end in states n and 0, respectively, then walk W' must end in the state $(n, 0)$, For the "only if" part, if walk W' ends in $(n, 0)$, then the walk for coordinate i cannot end in 0. Otherwise, if the walk for coordinate i ended in 0, then walk W' would reach a state $(0, y)$, for $0 < y < n$, and then end in either $(0, 0)$ or $(0, n)$, or would reach a state $(x, 0)$, for $0 < x < n$ and then end in $(0, 0)$, or would reach a state (x, n), for $0 < x < n$, and then end in $(0, n)$. Analogously, if walk W' ends in $(n, 0)$, then the walk for coordinate j cannot end in n.

For an unbiased random walk on $\{0, 1, \ldots, n\}$ staring at position X, the probability that the walk ends in the state 0 is equal to $(n - X)/n$. The one-dimensional random walks for the coordinates i and j start at positions i_0 and j_0, respectively. Thus the probability that the first walk ends in n is equal to $i_0/n = a$ and the probability that the second walk ends in 0 is equal to $(n - j_0)/n = 1 - b$.

5 Synchronous incremental voting on K_n: Theorem 1

In this section, we show Theorem 1, which refers to the synchronous process on the complete graph K_n. At each discrete time step, each vertex v chooses a vertex w independently and uniformly at random, and updates its opinion X_v to X'_v as in (1). We are interested in the evolution of $(X(t))_{t\geq 0}$.

Firstly, we show that the smallest opinion $s = \min_{v\in V} X_v(0)$ or the largest opinion $\ell = \max_{v\in V} X_v(0)$ vanishes w.h.p. within $O(\log n)$ steps, while $s + 3 \leq \ell$ (Lemma 10). Hence, after the smallest or largest opinion disappears $k - 3$ times, which occurs w.h.p. in $T = O(k \log n) = o(n/\log n)$ steps, at most three consecutive opinions $\{i - 1, i, i + 1\}$ are left. Using a Martingale concentration argument (Lemma 12) we further show that w.h.p. $|S(T) - S(0)| = O(\sqrt{nT \log n}) = o(n)$.

At this point only three adjacent values $\{i - 1, i, i + 1\}$ remain. In Lemma 15, we next either reduce the number of remaining opinions to two consecutive opinions, or if not, and we still have three opinions, then the sizes of opinions $i - 1$ and $i + 1$ are $o(n)$. This reduction takes $o(n)$ steps w.h.p., so we still have $|S(T) - S(0)| = o(n)$). In either case, the next, final phase completes in $O(n)$ expected steps, by comparing with pull voting. The comparison is straightforward, if only two consecutive opinions i and $i + 1$ remain.

When there are three opinions $i-1, i, i+1$, where $|A_{i-1} \cup A_{i+1}| = o(n)$, then $S(t)/n \sim i$, and we prove that w.h.p. i wins by coupling the process with pull voting. Let $A = A_i =$ and $B = A_{i-1} \cup A_{i+1}$. In pull voting value i wins with probability $|A|/n = 1 - o(1)$. After the first step of synchronous pull voting, $|A'_P| = Bin(n, |A|/n)$.

There is a coupling between incremental voting on three values, and pull voting such that $|A'_I|$ stochastically dominates $|A'_P|$. Firstly the number of vertices which choose in A directly is $Bin(n, |A|/n)$. Denote this set by $A' = A'_P$ and let $|A'_P| = X_P$. Given the set A'_P, a further non-negative number Y_I of vertices take the value i indirectly. This number Y_I is a sum of two binomials: $Y_I = Bin(|A_{i+1} \setminus A'_P|, N_{i-1}/n) + Bin(|A_{i-1} \setminus A'_P|, N_{i+1}/n)$. We have $A'_P \subseteq A'_I$, and the coupling can be extended to subsequent steps. Thus the probability that i wins in incremental voting is at least the probability that i wins in pull voting, so $1 - o(1)$.

5.1 Many opinions case

First, we show that one of the extreme opinions disappears within $O(\log n)$ steps.

▶ **Lemma 10.** *Let $s = \min_{v\in V} X_v(0)$ and $\ell = \max_{v\in V} X_v(0)$ be the smallest and the largest opinions in the initial round, respectively. Suppose $\ell \geq s + 3$. Then, $N_s(T)N_\ell(T) = 0$ w.h.p. within $T = O(\log n)$ steps.*

Applying Lemma 10 repeatedly, we immediately have the following.

▶ **Theorem 11.** *From any initial configuration of opinions from $[k] = \{1, 2, \ldots, k\}$, $X_v(T) \in \{i-1, i, i+1\}$ holds for some $1 < i < k$ and for any $v \in V$ within $T = O(k \log n)$ steps w.h.p.*

Proof of Lemma 10. By definition, we have that $N'_s \sim Bin\,(N_s + N_{s+1}, N_s/n)$ and $N'_\ell \sim Bin\,(N_{\ell-1} + N_\ell, N_\ell/n)$. Furthermore, N'_s and N'_ℓ are independent since $s+1 < \ell - 1$. Write $Z = N_s N_\ell$ and $Z' = N'_s N'_\ell$. Then, we have

$$\begin{aligned}\mathbf{E}\left[Z'\right] &= \mathbf{E}\left[N'_s N'_\ell\right] = \mathbf{E}\left[N'_s\right]\mathbf{E}\left[N'_\ell\right] = (N_s + N_{s+1})\frac{N_s}{n}(N_{\ell-1} + N_\ell)\frac{N_\ell}{n} \\ &= Z\frac{N_s + N_{s+1}}{n}\frac{N_{\ell-1} + N_\ell}{n} \le Z\frac{N_s + N_{s+1}}{n}\left(1 - \frac{N_s + N_{s+1}}{n}\right) \le \frac{1}{4}Z. \end{aligned} \tag{16}$$

The first inequality follows from $N_s + N_{s+1} + N_{\ell-1} + N_\ell \le n$. For $Z(t) = N_s(t)N_\ell(t)$, (16) implies that $\mathbf{E}\left[Z(t+1)\right] \le \mathbf{E}[Z(t)]/4$ holds for any $t \ge 0$. Taking $T = \lceil 3\log n \rceil$ and using the Markov inequality, we obtain

$$\mathbb{P}\left[Z(T) > 0\right] \le \mathbf{E}\left[Z(T)\right] \le \frac{1}{4}\mathbf{E}\left[Z(T-1)\right] \le \cdots \le \frac{1}{4^T}\mathbf{E}[Z(0)] \le \frac{n^2}{\mathrm{e}^{\lceil 3\log n\rceil}} \le \frac{1}{n}. \qquad \blacktriangleleft$$

5.2 Difference from the initial average

Next, we show that the average of the opinions is concentrated around the initial average.

▶ **Lemma 12.** *Let $S(t) = \sum_{v\in V} X_v(t)$. For any $T \ge 0$ and $\epsilon > 0$,*

$$\mathbb{P}\left[|S(T) - S(0)| \ge \epsilon\right] \le 2\exp\left(-\frac{\epsilon^2}{2nT}\right).$$

Proof. First, by Lemma 4 we observe that $(S(t))_{t=0,1,2,\ldots}$ is a martingale. From definition, we have $X_v(t+1) - X_v(t) \in \{-1, 0, 1\}$. Furthermore, for any $v \ne v'$, $X_v(t+1) - X_v(t)$ and $X_{v'}(t+1) - X_{v'}(t)$ are independent. Write $\Delta_v(t) = X_v(t) - X_v(t-1)$. Applying Lemma 19, we have

$$\begin{aligned}\mathbf{E}\left[\mathrm{e}^{\lambda(S(t+1)-S(t))}\middle|X(t)\right] &= \mathbf{E}\left[\mathrm{e}^{\lambda((S(t+1)-S(t))-\mathbf{E}[S(t+1)-S(t)|X(t)])}\middle|X(t)\right] \\ &= \mathbf{E}\left[\mathrm{e}^{\lambda\sum_{v\in V}(\Delta_v(t+1)-\mathbf{E}[\Delta_v(t+1)|X(t)])}\middle|X(t)\right] \\ &= \prod_{v\in V}\mathbf{E}\left[\mathrm{e}^{\lambda(\Delta_v(t+1)-\mathbf{E}[\Delta_v(t+1)|X(t)])}\middle|X(t)\right] \\ &\le \prod_{v\in V}\mathrm{e}^{\frac{\lambda^2}{2}} = \mathrm{e}^{\frac{\lambda^2 n}{2}}. \end{aligned} \tag{17}$$

Combining (17) and Lemma 20, we obtain the claim. $\blacktriangleleft$

▶ Remark. Choose $T = \lceil 3\log n\rceil$ from the proof of Theorem 11 and $\epsilon = \sqrt{7n}\log n$ in Lemma 12 to obtain

$$\mathbb{P}(|S(T) - S(0)| \ge \sqrt{7n}\log n) < \frac{1}{n}.$$

Note that the $2\exp\left(-\frac{\epsilon^2}{2nT}\right)$ bound in Lemma 12 is better than the $2\exp\left(-\frac{\epsilon^2}{2n^2T}\right)$ bound obtained directly from the Azuma-Hoeffding inequality (Lemma 18).

▶ **Lemma 13.** *Consider a synchronous vertex process on an arbitrary graph. Let $Z(t) = n\sum_{v\in V}\pi_v X_v(t)$. Then, for any $T \ge 0$ and $\epsilon > 0$,*

$$\mathbb{P}\left[|Z(T) - Z(0)| \ge \epsilon\right] \le 2\exp\left(-\frac{\epsilon^2}{2n^2\|\pi\|_2^2 T}\right),$$

where $\|\pi\|_2 = \sqrt{\sum_{v\in V}\pi_v^2}$.

Proof. First, by Lemma 4 we observe that $(Z(t))_{t=0,1,2,\ldots}$ is a martingale. From definition, we have $X_v(t+1) - X_v(t) \in \{-1, 0, 1\}$. Furthermore, for any $v \neq v'$, $X_v(t+1) - X_v(t)$ and $X_{v'}(t+1) - X_{v'}(t)$ are independent. Write $\Delta_v(t) = X_v(t) - X_v(t-1)$. Applying Lemma 19, we have

$$\begin{aligned}
\mathbf{E}\left[\mathrm{e}^{\lambda(Z(t+1)-Z(t))}\middle|X(t)\right] &= \mathbf{E}\left[\mathrm{e}^{\lambda((Z(t+1)-Z(t))-\mathbf{E}[Z(t+1)-Z(t)|X(t)])}\middle|X(t)\right] \\
&= \mathbf{E}\left[\mathrm{e}^{\lambda n\sum_{v\in V}\pi_v(\Delta_v(t+1)-\mathbf{E}[\Delta_v(t+1)|X(t)])}\middle|X(t)\right] \\
&= \prod_{v\in V}\mathbf{E}\left[\mathrm{e}^{\lambda n\pi_v(\Delta_v(t+1)-\mathbf{E}[\Delta_v(t+1)|X(t)])}\middle|X(t)\right] \\
&\leq \prod_{v\in V}\mathrm{e}^{\frac{4\lambda^2n^2\pi_v^2}{8}} = \mathrm{e}^{\frac{\lambda^2n^2\|\pi\|_2^2}{2}}. \qquad (18)
\end{aligned}$$

Combining (18) and Lemma 20, we obtain the claim. ◀

For regular graphs, both π_v and $\|\pi\|_2^2$ are $1/n$. So Lemma 13 generalizes Lemma 12.

5.3 At most three consecutive opinions remain

In this section, we suppose that $X_v(0) \in \{i-1, i, i+1\}$ holds for some i and for all $v \in V$, i.e., all initial opinions are from three consecutive integers. Without loss of generality, we assume that $i = 2$ throughout this section.

▶ **Lemma 14.** *Suppose that $X_v(0) \in \{1, 2, 3\}$ holds for all $v \in V$. Then, for any $t \geq 0$,*

$$\mathbf{E}[N_1(t+1)N_3(t+1) \mid X(t)] \leq \left(1 - \frac{N_1(t) + N_3(t)}{2n}\right) N_1(t)N_3(t).$$

Proof. Let $Y_{i\to j}$ denote the number of vertices that change their opinion from i to j. We have $N_1' = Y_{1\to1} + Y_{2\to1}$ and $N_3' = Y_{2\to3} + Y_{3\to3}$. Note that $Y_{3\to1} = Y_{1\to3} = 0$. It is easy to see that $Y_{1\to1} \sim Bin(N_1, N_1/n)$ and $Y_{3\to3} \sim Bin(N_3, N_3/n)$. An important observation is that $(Y_{2\to1}, Y_{2\to2}, Y_{2\to3})$ follows a multinomial distribution with parameters N_2 and $(N_1/n, N_2/n, N_3/n)$. Hence, $Cov(Y_{2\to1}, Y_{2\to3}) \leq 0$ and we have $\mathbf{E}[Y_{2\to1}, Y_{2\to3}] \leq \mathbf{E}[Y_{2\to1}]\mathbf{E}[Y_{2\to3}]$. Thus,

$$\begin{aligned}
\mathbf{E}[N_1'N_3'] &= \mathbf{E}[Y_{1\to1}(Y_{2\to3} + Y_{3\to3})] + \mathbf{E}[Y_{2\to1}Y_{2\to3}] + \mathbf{E}[Y_{2\to1}Y_{3\to3}] \\
&\leq \mathbf{E}[Y_{1\to1}]\mathbf{E}[Y_{2\to3} + Y_{3\to3}] + \mathbf{E}[Y_{2\to1}]\mathbf{E}[Y_{2\to3}] + \mathbf{E}[Y_{2\to1}]\mathbf{E}[Y_{3\to3}] \\
&= (\mathbf{E}[Y_{1\to1}] + \mathbf{E}[Y_{2\to1}])(\mathbf{E}[Y_{2\to3}] + \mathbf{E}[Y_{3\to3}]) \\
&= \left(N_1\frac{N_1}{n} + N_2\frac{N_1}{n}\right)\left(N_2\frac{N_3}{n} + N_3\frac{N_3}{n}\right) \\
&= N_1\left(1 - \frac{N_3}{n}\right)N_3\left(1 - \frac{N_1}{n}\right). \qquad (19)
\end{aligned}$$

Note that $Y_{i\to j}$ and $Y_{k\to\ell}$ are independent for $i \neq k$. Combining (19) and the fact that $(1-x)(1-y) = 1 - x - y + xy \leq 1 - (x+y) + \frac{(x+y)^2}{2} \leq 1 - \frac{x+y}{2}$ holds for any $0 \leq x + y \leq 1$, we obtain the claim. ◀

Intuitively speaking, Lemma 14 implies that $N_1(t)N_3(t)$ continues to decrease by a factor of $1 - 1/\sqrt{n}$ while $N_1(t) + N_3(t) \geq 2\sqrt{n}$. Hence, within $T = O(\sqrt{n}\log n)$ steps, $N_1(t)N_3(t)$ reaches 0 or $N_1(t) + N_3(t) < 2\sqrt{n}$. In other words, either of the following events occurs: (1) either $N_1(t)$ or $N_3(t)$ is zero, (2) both $N_1(t)$ and $N_3(t)$ are less than $2\sqrt{n}$. The following lemma shows it formally.

▶ **Lemma 15.** *Suppose that $X_v(0) \in \{1,2,3\}$ holds for all $v \in V$. Let $T = \lceil 3\sqrt{n}\log n \rceil$. Then, for some $0 \le t \le T$, w.h.p. one of the following two events occurs.*

1. *$N_1(t) = 0$ or $N_3(t) = 0$.*
2. *$N_1(t) \le 2\sqrt{n}$ and $N_3(t) \le 2\sqrt{n}$.*

Proof. Let

$$\tau = \min\{t \ge 0 \mid N_1(t) + N_3(t) < 2\sqrt{n} \text{ or } N_1(t)N_3(t) = 0\},$$
$$Y_t = N_1(t)N_3(t)\left(1 - \frac{1}{\sqrt{n}}\right)^{-t}, \quad Z_t = Y_{t\wedge\tau} = Y_{\min\{t,\tau\}}.$$

Note that we have $Z_{t+1} - Z_t = \mathbb{1}_{\tau>t}(Y_{t+1} - Y_t)$. From Lemma 14,

$$\begin{aligned}\mathbf{E}[Z_{t+1} - Z_t \mid X(t)] &= \mathbb{1}_{\tau>t}\left(\mathbf{E}[Y_{t+1} \mid X(t)] - Y_t\right)\\ &\le \mathbb{1}_{\tau>t}\left(\frac{\left(1 - \frac{N_1(t)+N_3(t)}{2n}\right)N_1(t)N_3(t)}{\left(1-\frac{1}{\sqrt{n}}\right)^{t+1}} - \frac{N_1(t)N_3(t)}{\left(1-\frac{1}{\sqrt{n}}\right)^{t}}\right)\\ &\le 0\end{aligned}$$

holds, i.e., $\mathbf{E}[Z_{t+1}] \le \mathbf{E}[Z_t]$ and $\mathbf{E}[Z_T] \le \mathbf{E}[Z_0] = Z_0$. Let $T = \lceil 3\sqrt{n}\log n\rceil$. Then,

$$\mathbf{E}[Z_T \mid \tau > T]\mathbb{P}[\tau > T] \le \mathbf{E}[Z_T] \le Z_0 = N_1(0)N_3(0) \le n^2. \tag{20}$$

Furthermore,

$$\mathbf{E}[Z_T \mid \tau > T] = \mathbf{E}\left[N_1(T)N_3(T)\left(1-\frac{1}{\sqrt{n}}\right)^{-T} \mid \tau > T\right] \ge \left(1-\frac{1}{\sqrt{n}}\right)^{-3\sqrt{n}\log n} \ge n^3. \tag{21}$$

Note that the event $\tau > T$ implies that $N_1(T) + N_3(T) \ge 2\sqrt{n}$ and $N_1(T)N_3(T) \ge 1$. Combining Equations (20) and (21), we obtain $\mathbb{P}[\tau > T] \le 1/n$. ◀

Completing the proof of Theorem 1. If we have reached here at some step t, then at most three values $i-1, i, i+1$ remain, and one of the cases Lemma 15 (1) or Lemma 15 (2) holds. We next prove that w.h.p. the process will finish in $O(n)$ steps with the claimed results.

In either case, by Lemma 12 $S(t) = S(0)(1 + o(1))$. So if Lemma 15 (1) holds, there are two remaining values, say $i, i+1$, and we can use two-value pull voting with Theorem 5 directly.

However if Lemma 15 (2) holds, then there are three values $i-1, i, i+1$, where $|A_{i-1} \cup A_{i+1}| = O(n^{1/2})$. Thus $S(t)/n \sim i$, and we next prove that i wins w.h.p. by coupling the process with pull voting. For convenience let $\{i-1, i, i+1\} = \{1,2,3\}$, let $A_2 = A$ and $B = A_1 \cup A_3$. In pull voting value $i = 2$ wins with probability $|A|/n = 1 - o(1)$. In one step of synchronous pull voting, $|A'_P| = Bin(n, |A|/n)$.

There is a coupling between incremental voting on three values, and pull voting such that $|A'_I|$ stochastically dominates $|A'_P|$. Firstly the number of vertices which choose in $A = A_2$ directly is $Bin(n, |A|/n)$. Denote this set by $A' = A'_P$ and let $|A'_P| = X_P$. Given the set A'_P, a further non-negative number Y_I of vertices take the value $i = 2$ indirectly. The value of Y_I is a sum of binomials, namely

$$Y_I = Bin(|A_3 \setminus A'_P|, N_1/n) + Bin(|A_1 \setminus A'_P|, N_3/n).$$

It follows that under the coupling $|A'_I| = X_P + Y_I \ge X_P = |A'_P|$ and thus

$$\mathbb{P}(\text{Value } i \text{ wins in incremental voting}) \ge \mathbb{P}(\text{Value } i \text{ wins in pull voting}) = 1 - o(1).$$

6 Concluding comments

The incremental voting model offers an alternative type of pull voting suitable for discrete numeric opinions which can be compared on a linear scale. This may be appropriate for systems which need a very simple protocol which converges towards an average opinion. As the extremal values are discarded rapidly in some instances, it could also offer a faster alternative to remove outliers in some plurality systems.

The incremental voting process can be viewed as a form of discrete averaging of integer weights. The final answer is an integer (no fractions), obtained in finite expected time. For suitable expanders, w.h.p. the process returns the average rounded up or down to an integer. To increase the accuracy of the averaging, multiply all initial values by 10^h before averaging. The final answer, after re-scaling, will now be w.h.p. correct to the h-th decimal place. The cost is the increased convergence time.

In incremental voting, the weighted average remains a martingale under a wide range of conditions. Let P be any reversible transition matrix and π be its stationary distribution. Then, if the selected vertex v chooses u with probability $P(v,u)$, the random variable $W = \sum_{v \in V} \pi_v X_v$ is a martingale. As an example, if $P(u,u) = 1 - L_u$ and $P(u,v) = L_u/d(u)$, where $0 < L_u \leq 1$, then L_u can be viewed as the propensity for vertex u to change its opinion when selected in a given step. Here, $\pi(v) = d(v)/CL_v$, where $C = \sum L_v/d(v)$.

References

1 Mohammed Amin Abdullah and Moez Draief. Global majority consensus by local majority polling on graphs of a given degree sequence. *Discrete Applied Mathematics*, 180:1–10, 2015. doi:10.1016/J.DAM.2014.07.026.

2 David Aldous and James Allen Fill. Reversible markov chains and random walks on graphs. Unfinished monograph (recompiled version, 2014), 2002.

3 Luca Becchetti, Andrea Clementi, Emanuele Natale, Francesco Pasquale, and Riccardo Silvestri. Plurality consensus in the gossip model. In *Proceedings, 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 371–390, Philadelphia, PA, 2015. SIAM. doi:10.1137/1.9781611973730.27.

4 Luca Becchetti, Andrea Clementi, Emanuele Natale, Francesco Pasquale, Riccardo Silvestri, and Luca Trevisan. Simple dynamics for plurality consensus. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 247–256, New York, NY, USA, 2014. ACM. doi:10.1145/2612669.2612677.

5 Luca Becchetti, Andrea Clementi, Emanuele Natale, Francesco Pasquale, and Luca Trevisan. Stabilizing consensus with many opinions. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 620–635, Philadelphia, PA, 2016. SIAM. doi:10.1137/1.9781611974331.CH46.

6 Petra Berenbrink, George Giakkoupis, and Peter Kling. Tight bounds for coalescing-branching random walks on regular graphs. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1715–1733, Philadelphia, PA, 2018. SIAM. doi:10.1137/1.9781611975031.112.

7 Siddhartha Brahma, Sandeep Macharla, Sudebkumar Prasant Pal, and Sudhir Kumar Singh. Fair leader election by randomized voting. In *Proceedings of the 1st international conference on Distributed Computing and Internet Technology (ICDCIT)*, pages 22–31. Springer, 2004. doi:10.1007/978-3-540-30555-2_4.

8 Colin Cooper, R. Elsässer, and Tomasz Radzik. The power of two choices in distributed voting. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 435–446. Springer, 2014. doi:10.1007/978-3-662-43951-7_37.

9 Colin Cooper, Robert Elsässer, Hirotaka Ono, and Tomasz Radzik. Coalescing random walks and voting on graphs. In *Proceedings of the 31st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 47–56, New York, NY, USA, 2012. ACM. doi:10.1145/2332432.2332440.

10 Colin Cooper, Tomasz Radzik, Nicolás Rivera, and Takeharu Shiraga. Fast plurality consensus in regular expanders. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, volume 91, pages 13:1–13:16. Springer, 2017. doi:10.4230/LIPICS.DISC.2017.13.

11 Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. Stabilizing consensus with the power of two choices. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 149–158, New York, NY, USA, 2011. ACM. doi:10.1145/1989493.1989516.

12 Mohsen Ghaffari and Johannes Lengler. Nearly-tight analysis for 2-choice and 3-majority consensus dynamics. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 305–313, New York, NY, USA, 2018. ACM. URL: https://dl.acm.org/citation.cfm?id=3212738.

13 Yehuda Hassin and David Peleg. Distributed probabilistic polling and applications to proportionate agreement. *Information and Computation*, 171(2):248–268, 2001. doi:10.1006/INCO.2001.3088.

14 Barry Johnson. *Design and analysis of fault tolerant digital systems*. Addison-Wesley, Boston, MA, USA, 1989.

15 Nan Kang and Nicolás Rivera. Best-of-three voting on dense graphs. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 115–121, New York, NY, USA, 2019. ACM. doi:10.1145/3323165.3323207.

16 Toshio Nakata, Hiroshi Imahayashi, and Masafumi Yamashita. Probabilistic local majority voting for the agreement problem on finite graph. In *Proceedings of the 5th Annual International Computing and Combinatorics Conference (COCOON)*, pages 330–338. Springer, 1999. doi:10.1007/3-540-48686-0_33.

17 Roberto Imbuzeiro Oliveira and Yuval Peres. Random walks on graphs: new bounds on hitting, meeting, coalescing and returning. In *Proceedings of the 16th Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 119–126, Philadelphia, PA, 2019. SIAM. doi:10.1137/1.9781611975505.13.

18 Nobutaka Shimizu and Takeharu Shiraga. Phase transitions of best-of-two and best-of-three on stochastic block models. *Random Structures and algorithms*, 59(1):96–140, 2021. doi:10.1002/RSA.20992.

A Proof of Lemma 8

We repeat the lemma for convenience.

▶ **Lemma 16** (Lemma 8). *Let $G \in G_{n,p}$, where $np \geq \log^{1+\varepsilon} n$ for some constant $\varepsilon > 0$. The following properties hold w.h.p..*

P1. (Almost regular graphs) *G is connected and all vertices v have degree $d(v) = np + O(\sqrt{np \log n})$ and stationary distribution $\pi_v = \frac{1}{n} + O\left(\frac{1}{n \log^{\varepsilon/2} n}\right)$.*

P2. (Large number of edges between large subsets of vertices)
Let $\delta \geq 5/\sqrt{np}$. For any pair of disjoint vertex sets A, B, with $|A| \geq \delta n$, $|B| \geq \delta n$, the number of edges X_{AB} between A and B satisfies $\mu/2 \leq X_{AB} \leq 3\mu/2$, where $\mu = |A||B|p$, the expected number of edges between the sets A and B in $G_{n,p}$.

P3. (Not too many edges within small subsets of vertices)

(i) *For $\omega \geq e$, $\omega \log \omega \leq np$, no vertex set S of size $s = n/\omega$ induces more than $X_S = e^2 s^2 p$ edges.*

(ii) *Let $d = np$. No set S, $|S| \le n/\omega$ induces more than $X_S = s\sqrt{4d\log(ne/s)}$ edges.*
(iii) *Provided $\omega = O(\log n)$ and $np = d \ge \log^{1+\theta} n$, the ratio $X_S/X_{S,V-S}$ is $O(1/\omega)$, the value achieved in P3.(i) above.*

Proof.

P1. An application of the Chernoff-Hoeffding inequality (Lemma 17) shows that for all vertices v, $d(v) = np + O(\sqrt{np\log n})$.

P2. for given disjoint A, B Let $|A| = an$, $|B| = bn$ then the Chernoff-Hoeffding inequality (Lemma 17.3) with $\varepsilon = 1/2$ implies

$$P_{AB} = \mathbb{P}(X_{AB} \notin [\mu/2, 3\mu/2]) \le 2e^{-\mu/12}.$$

We say that A, B is a *bad pair*, if $|A| \ge \delta n$ and $|B| \ge \delta n$ but $X_{AB} \notin [\mu/2, 3\mu/2]$. Then

$$\mathbf{E}(\text{number of bad pairs}) = \sum_{A,B} P_{AB} \le$$
$$\le 4^n\, 2e^{-\delta^2 n^2 p/12} \le 2\left(4e^{-\delta^2 np/12}\right)^n \le 2\left(4e^{-2}\right)^n = o(1).$$

P3. (i) Let X_S denote the number of edges induced by a set S, and $\mu = \mathbf{E}(X_S) = \binom{s}{2}p$ the expected number. By the Chernoff-Hoeffding inequality (Lemma 17.4), for $\alpha \ge e$ and $s \ge 3$,

$$P_S = \mathbb{P}(X_S \ge \alpha\mu) \le (e/\alpha)^{\alpha\mu} \le (e/\alpha)^{\alpha s^2 p/3}. \tag{22}$$

Say a set S of size s is a *bad set*, if it induces more than $e^2 s^2 p \ge e^2\mu$ edges. Then, using (22) with $\alpha = e^2$,

$$\begin{aligned}\mathbf{E}(\text{number of bad sets of size } s) &\le \binom{n}{s} e^{-e^2 s^2 p/3} \\ &\le \left(\frac{ne}{s} e^{-e^2 sp/3}\right)^s = \left(\exp\{-e^2 sp/3 + \log ne/s\}\right)^s \\ &\le \left(e^{-(e^2/3)\log\omega + \log\omega e}\right)^s \le \left(e^{-(e^2/3-2)\log\omega}\right)^s = o(1).\end{aligned}$$

The last inequality follows from the assumption that $\omega \ge e$. The size $s = n/\omega$ is minimized when $\omega\log\omega = np$, implying that $s \ge (\log\log n)/2$. Sum the above over all s greater than this minimum value to conclude that the expected number of bad sets of sizes in the required range is $o(1)$.

(ii) Let $\mu = \mathbf{E}X_{S,V-S} = s(n-s)p$. Then, as $n - s = n(1 - o(1))$, $\mu = sd(1 - o(1))$ and

$$P_{S,V-S} = \mathbb{P}(X_{S,V-S} \le (1-\varepsilon)\mu) \le e^{-\varepsilon^2 sd/3}.$$

Thus

$$\mathbf{E}(\text{number of bad sets } S) \le \binom{n}{s} P_{S,V-S} \le \left(\frac{ne}{s} e^{-\varepsilon^2 d/3}\right)^s = o(1),$$

provided $\varepsilon \ge \sqrt{\frac{4\log(ne/s)}{d}}$.

As the total degree of S is $sd(1 + 0(1))$, no such S can induce as many as

$$X_S = \varepsilon s(n-s)p \le \varepsilon sd \le s\sqrt{4d\log(ne/s)}$$

edges.

(iii) Thus for $s \le n/\omega$,

$$\max \frac{X_S}{X_{S,V-S}} = O\left(\frac{s\sqrt{d\log n/s}}{sd}\right) = O\left(\sqrt{\frac{\log n}{d}}\right) = O\left(\frac{1}{\log^\theta n}\right) = O\left(\frac{1}{\omega}\right),$$

provided $\omega = O(\log n)$, and $np = d \ge \log^{1+\theta} n$. ◀

B Tools used in the analysis

▶ **Lemma 17** (The Chernoff-Hoeffding inequalities). *Let $X_1, \ldots, X_n$ be n independent random variables taking values in $[0,1]$. Let $X = \sum_{i=1}^{n} X_i$. Let $\mu^- \le \mathbf{E}[X] \le \mu^+$. Then, we have the following:*

1. $\mathbb{P}[X \ge (1+\varepsilon)\mu^+] \le \exp\left(-\frac{\min\{\varepsilon^2, \varepsilon\}\mu^+}{3}\right)$, *for* $\varepsilon \ge 0$.
2. $\mathbb{P}[X \le (1-\varepsilon)\mu^-] \le \exp\left(-\frac{\varepsilon^2\mu^-}{2}\right)$, *for* $0 \le \varepsilon \le 1$.
3. $\mathbb{P}[X \notin ((1-\varepsilon)\mu^-, (1+\varepsilon)\mu^+)] \le 2\exp\left(-\frac{\varepsilon^2\mu^-}{3}\right)$, *for* $0 \le \varepsilon \le 1$.
4. $\mathbb{P}[X \ge \alpha\mu^+] \le \left(\frac{e^{\alpha-1}}{\alpha^\alpha}\right)^{\mu}$, *for* $\alpha \ge 1$.

▶ **Lemma 18** (The Azuma-Hoeffding inequality). *Let $(X_t)_{t=0,1,2,\ldots}$ be a martingale. Suppose $|X_i - X_{i-1}| \le c_i$ holds for any $i \ge 0$. Then, for any $T \ge 0$ and $\epsilon > 0$,*

$$\mathbb{P}[|X_T - X_0| \ge \epsilon] \le 2\exp\left(-\frac{\epsilon^2}{2\sum_{i=1}^{T} c_i^2}\right).$$

The followings are the basic technical lemmas for the Hoeffding inequality.

▶ **Lemma 19.** *Let X be a random variable such that $\mathbf{E}[X] = 0$ and $a \le X \le b$. Then, for any $\lambda > 0$, $\mathbf{E}\left[e^{\lambda X}\right] \le e^{\lambda^2(b-a)^2/8}$.*

▶ **Lemma 20.** *For any $\alpha > 0$ and t, suppose that $\mathbf{E}\left[e^{\alpha(Y_t - Y_{t-1})} \mid \mathcal{F}_{t-1}\right] \le e^{\alpha^2 c_t^2}$ holds for some c_t. Then, for any $\epsilon > 0$, $\mathbb{P}[|Y_T - Y_0| \ge \epsilon] \le 2\exp\left(-\frac{\epsilon^2}{4\sum_{t=1}^{T} c_t^2}\right)$.*

On the Convergence Time in Graphical Games: A Locality-Sensitive Approach

Juho Hirvonen
Aalto University, Finland
Helsinki Institute for Information Technology (HIIT), Espoo, Finland

Laura Schmid
Kim Jaechul Graduate School of AI, KAIST, Seoul, Republic of Korea

Krishnendu Chatterjee
IST Austria, Klosterneuburg, Austria

Stefan Schmid
TU Berlin, Germany
Weizenbaum Institute, Berlin, Germany

Abstract

Graphical games are a useful framework for modeling the interactions of (selfish) agents who are connected via an underlying topology and whose behaviors influence each other. They have wide applications ranging from computer science to economics and biology. Yet, even though an agent's payoff only depends on the actions of their direct neighbors in graphical games, computing the Nash equilibria and making statements about the convergence time of "natural" local dynamics in particular can be highly challenging. In this work, we present a novel approach for classifying complexity of Nash equilibria in graphical games by establishing a connection to local graph algorithms, a subfield of distributed computing. In particular, we make the observation that the equilibria of graphical games are equivalent to locally verifiable labelings (LVL) in graphs; vertex labelings which are verifiable with constant-round local algorithms. This connection allows us to derive novel lower bounds on the convergence time to equilibrium of best-response dynamics in graphical games. Since we establish that distributed convergence can sometimes be provably slow, we also introduce and give bounds on an intuitive notion of "time-constrained" inefficiency of best responses. We exemplify how our results can be used in the implementation of mechanisms that ensure convergence of best responses to a Nash equilibrium. Our results thus also give insight into the convergence of strategy-proof algorithms for graphical games, which is still not well understood.

2012 ACM Subject Classification Theory of computation → Network games; Theory of computation → Algorithmic game theory

Keywords and phrases distributed computing, Nash equilibria, mechanism design, best-response dynamics

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.11

Related Version *Full Version*: `https://arxiv.org/abs/2102.13457`

Funding This work was partially funded by the Academy of Finland, grant 314888, the European Research Council CoG 863818 (ForM-SMArt), and the Austrian Science Fund (FWF) project I 4800-N (ADVISE). LS was supported by the Stochastic Analysis and Application Research Center (SAARC) under National Research Foundation of Korea grant NRF-2019R1A5A1028324.

Acknowledgements We thank the anonymous reviewers for their feedback.

1 Introduction

Modeling the interactions of multiple selfish agents, whose decisions and behavior influence each other and are in some way dependent on an underlying topology, is an important aspect of solving problems from a wide range of diverse fields. Game theory offers a natural approach to this issue in the form of multiplayer network games [28, 9]. Instances of such

network games are ubiquitous [3, 40, 28, 2, 43, 1]. They all have in common that interactions and players' decisions are in some way governed by the underlying graph and that players' payoffs usually depend on the network that links them. Considering their heterogeneity and complexity, general network games, such as the rich class of congestion games [40], can be difficult to analyze. A considerable amount of research has thus been dedicated to more restricted versions of such games, such as those where each player's utility is solely determined by their own and their direct neighbors' actions. These so-called *graphical games* (see [31] for a comprehensive introduction and survey) capture the *locality* of effects on players, and are useful for settings where there are only a few strong direct influences on every agent [9].

Computing the Nash equilibria of graphical games or proving results about their properties or convergence is challenging in general. Previous work has dealt with aspects such as deciding whether a given strategy profile is a Nash equilibrium or whether equilibria exist, and classifying the complexity of these tasks [41]. Algorithms for computing (approximate) equilibria, often for various specific types of graphs, have also been a focus of a considerable number of studies [37, 20, 21, 44, 27, 22]; however, it is important to note that these algorithms usually do not reflect natural game dynamics. Hence, they are not guaranteed to be efficient or strategy-proof: players can have incentives to deviate from the algorithm. Additionally, we also still lack a systematic understanding and classification in terms of computational complexity. In particular, the convergence time of strategy-proof algorithms and local dynamics, such as best-response dynamics, to Nash equilibria is still not fully understood, except for special cases [32, 26, 42].

In this work, we approach graphical games from a new perspective, demonstrating that the theory of distributed computing is particularly well-suited as a tool to study the behavior and computational power of decentralized game-theoretic systems with multiple players. In particular, distributed computing offers the tools to prove unconditional and general impossibility results, which demonstrate natural limits to the behavior of systems. One of the most significant limiting factors in distributed computing is *locality* [39], or how far information must propagate in a system to solve a particular task. Advantageously, impossibility results based on locality then apply independently of more specific factors, such as the model of strategic behavior considered. This makes it possible to analyse games without a fixed model of play.

Distributed computing further provides a benchmark for optimal cooperation. By comparing the outcome of models such as best-response dynamics to the best efficient distributed algorithm, we can assess the inefficiency of systems that are not perfectly coordinated. We also get natural algorithmic tools for mechanism design: properly incentivised, distributed algorithms can potentially be run by the agents without the need for centralised control. We also note that the tools of distributed computing theory help us understand games with multiple equilibria. We can consider questions such as which subset of equilibria can be computed efficiently, and what their properties are. Additionally, we can study how likely *undesirable* behavior is, noting that bad equilibria are also often hard to compute.

Investigating graphical games as distributed systems may seem very natural, but has only been rarely explored [19]. In order to showcase the rich set of tools resulting from this perspective, we show how Nash equilibria and other stable state concepts in graphical games correspond to a family of computational problems in distributed computing called *locally verifiable labellings (LVLs)* [36]. The computational complexity of these problems is well studied (see e.g. [36, 16, 11, 17, 5]). Using the properties of LVLs, we can then classify graphical games based on how hard it is to compute a Nash equilibrium, i.e. based on the distributed hardness of computing a stable state. We show how this implies lower bounds

on the convergence time of best-response dynamics in network games: these dynamics can converge only as fast as a distributed algorithm can compute solutions to the equivalent graph problem. We find three types of graphical games in the process: (1) Games where Nash equilibria are easy to compute and best responses also converge fast; (2) games where computing a Nash equilibrium is hard, and thus best responses cannot converge fast; and (3) games where Nash equilibria are easy to compute, but best responses may fail to converge fast. We exemplify our approach with three corresponding fundamental graphical games that have been studied in the literature [8, 9, 30].

Games of the third type are particularly interesting, as the existence of fast distributed algorithms offers possibilities for mechanism design. We show a way of constructing games where best responses simulate distributed algorithms, thus ensuring convergence in games of type 3. Since best-response dynamics are rational with respect to the restricted local knowledge of agents, our work also connects to the open question of strategy-proof algorithms for distributed Nash equilibria computation that is posited by [29].

Finally, we define a notion called inefficiency of best responses: It is a time-constrained analogue of the Price of Anarchy that compares the evolution of the system state over a time period with the best solution that can be computed with the same time complexity. We demonstrate that Price of Anarchy can be an unfair measure by comparing strategic behavior with perfect *distributed* cooperation.

Overall, our approach of classifying games based on the computational hardness of their equilibria provides a powerful new perspective on graphical games that allows for a deeper understanding of strategic behavior, and contributes to a strengthened connection between game theory and distributed computing.

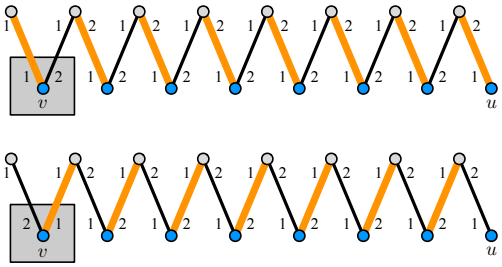


Figure 1 Two simple networks with preferences indicated by numbers: each agent prefers the neighbour labelled 1 over the neighbour labelled 2. The two networks only differ in the preferences of the agent labelled v. Still, the unique stable matchings (marked with bold edges) are disjoint.

Stable matching [24] is a classic example of a graphical game: we have a bipartite network where each agent on the left is connected to some subset of agents on the right. Agents have some preference relation over the other agents they are connected to, and the goal is to find a matching such that no pair of connected agents prefer each other over their current matches. While there is a polynomial-time truthful *centralised* mechanism for finding a stable matching, the problem becomes hard when agents form a decentralized distributed system [23]: in the worst case, the output of an agent depends on the preferences of other agents at linear distance (see Figure 1). This example highlights the goal of our work: we study games that are considered "easy" is the literature of game theory and provide a more fine-grained view of their hardness from the perspective of decentralised systems.

1.1 Organization of the paper

We begin with a primer on graphical games and locally verifiable labelings in Section 2. We proceed to show in Section 3 how Nash equilibria of graphical games correspond to a family of computational tasks called locally verifiable labellings. Then, we show how to apply these results to best-response game dynamics. In Section 4 we study and classify three fundamental graphical games. In Section 5 we study how distributed algorithms can be used to incentivize desired agent behavior. In Section 6, we use theory of distributed computing to study how inefficient best-response dynamics are compared to distributed algorithms. We conclude with an outlook on the connection between theory of distributed computing and game theory. To improve readability and provide intuition, we, as indicated, defer technical details and full proofs to the Appendix.

2 Model

2.1 Graphical games

Define a multiplayer game in its general form as consisting of n players $i \in \mathcal{I}$, each equipped with a pure strategy space or action space A_i. The cartesian product $A_1 \times A_2 \times ... \times A_n$ of the action spaces of individual agents is denoted by $\mathcal{A}$. Every player has a utility function $u_i(a)$ mapping each strategy profile to a value: $u_i \colon \mathcal{A} \times \mathcal{I} \to \mathbb{R}$. We use a_{-i} to denote the joint strategy profile of all players except for player i.

Throughout the paper, our examples will focus on pure strategies, but all of the general results also apply to mixed strategies. A strategy profile a^* is a (pure) Nash equilibrium if for all players i it holds that $u_i(a_i^*, a_{-i}^*) \geq u_i(a_i, a_{-i}^*) \ \ \forall a_i \in A_i$. In other words, no player in a Nash equilibrium can gain a higher utility by unilaterally deviating. We say that the strategy a_i^* is a *best response* to the rest of the strategy profile a_{-i}^*.

We can now define *graphical games*, given by a triple $(\mathcal{A}, u, N)$. As introduced in [31], graphical games are a concisely representable form of *multiplayer games* on networks. A *network* or a *graph* $N = (V, E)$ consists of a set of n *nodes* V and a set E of *edges* between pairs of nodes. The nodes of the network N then represent the agents, or players, in the graphical game. We define the local neighborhood of a node v as $B(v) \subseteq \{1, .., n\} = \{j \in V, (v, j) \in E\}$, with $v \in B(v)$ as well. A player's *utility function* $u_v(a)$ now depends only on a strategy profile restricted to their local neighborhood $B(v)$, i.e. the partial strategy profile $a^{(v)} = (a_i : i \in B(v))$. We denote the product of the individual agents' utility functions by u.

A classic example of graphical games are coloring games [30]. In a k-coloring game, the agents must choose resources (colors) that differ from the resources chosen by their neighbors. The action set $A_i = \{1, 2, \ldots, k\}$ for all agents. Utility is 1 if the strategy a_i is a color that is not chosen by its neighbors, and 0 otherwise. A strategy profile is a Nash equilibrium if none of the agents with a similarly colored neighbor can choose a free color.

In this work, we consider the question of *convergence* to Nash equilibria via *best-response dynamics*, a specific example of *local dynamics*. We assume that players can update their strategy in between gameplay. In one step of the dynamics, one player is selected and this player then updates her strategy with the best response to her neighborhood's strategy profile. That is, the action of a node v is a best response to the partial strategy profile $a_v(t-1)$. There is some fixed order on the actions that is used to break ties. We assume that nodes only have local information and cannot look beyond their neighborhood. Their restricted knowledge makes such local dynamics rational. In this work, we will subsequently only consider best responses. However, our results also hold for more general local dynamics.

In order to have a reasonable definition of a running time for local dynamics, we define a model of *fair best responses*. The play consists of *fair rounds*. During each round an adversary schedules all agents to act exactly once and one at a time. The convergence time of best-response dynamics on a fixed network is the maximum number of fair rounds until all players have reached a Nash equilibrium over, all possible orders of play. For random initial strategy profiles we say that best-response dynamics converge if they converge with high probability over the initial strategy assignment.

We choose the liveness criterion of each player playing each round for two reasons. First, if there is no bound on how often each player is guaranteed to play, then best responses will not converge, simply because the adversary can stop some fixed subset of players from playing. Second, the effect of e.g. each player playing every k rounds simply appears as a multiplicative factor in the convergence times.

2.2 Distributed complexity theory and locally verifiable labelings

Our work establishes a connection between graphical games and the *LOCAL* model of computation [35, 39]). We are given a fixed network $N = (V, E)$ (a graph) connecting $n = |V(N)|$ nodes. To identify the nodes, each node gets an $O(\log n)$-bit unique name as an input. We will also consider the randomized LOCAL model, where instead each node has access to its own private source of random bits. The nodes then collaboratively aim to solve a given task, but can only communicate with their neighbors. The computation proceeds in *synchronous* rounds. In each round each node in the graph can (1) send a message to each of its neighbors (there is no bound on the message size), (2) receive a message from each neighbor, and (3) perform local computations (there is no bound on the complexity of these local computations). Initially the nodes do not know anything about the input network. Each node is responsible for computing its own part of the output.

Running time is measured in the number of communication rounds until all nodes have stopped and announced their outputs. The *running time* of a distributed algorithm (over a family $\mathcal{N}$ of networks) is the maximum running time over all networks and inputs (identifiers, random bits). The *distributed complexity* of a problem P for a family $\mathcal{N}$ of networks is the running time of the optimal algorithm for solving P over $\mathcal{N}$.

We note that the crucial property of the LOCAL model is that in T communication rounds each node can gather information from only its T-hop neighborhood in the network. Information outside this radius cannot affect the actions of a node, due to requiring a longer time to propagate. We denote this T-hop neighborhood of a node v by $B(v, T)$, and the radius-1 neighborhood simply by $B(v)$.

We study a class of problems called *locally verifiable labelling (LVL)* problems. This is a generalization of *locally checkable labellings (LCLs)* [36], a family of problems that has been studied extensively in recent years (for example [11, 16, 17, 5, 14, 10, 4]). These are the class of problems that can be *verified* efficiently with a distributed algorithm: a solution is correct if and only if it is correct in the constant-radius neighborhood of each node. Conversely, if a solution is not correct, at least one node can detect this by looking at its constant-radius neighborhood. For example, coloring is locally verifiable, as each node can compare its color with its neighbors and verify that the colors differ.

As we will show in Section 3, LVLs also capture the Nash equilibria of graphical games (as computational problems). This follows naturally from the fact that the stability of each agent only depends on the strategies of its neighbors.

An important result in the theory of distributed computing is a "complexity gap" of LVLs [16]. It allows us to classify LVLs into *easy* and *hard* problems:

▶ **Fact 1** (Chang et al. [16], Informal). *There is a superexponential gap in the possible distributed complexities of LVL problems. Easy problems below this gap can be solved in almost constant time, and hard problems above this gap require logarithmic time.*

The running time for easy problems is at most the iterated logarithm of the size of the network – an extremely slowly growing function. We say that an algorithm with this running time is *fast*. Hard problems require at least logarithmic time in the size of the network. We call such running times *slow*. For more details, see Appendix A.2.

3 Distributed computing and graphical games

Our work is motivated by our observation that graphical games and local algorithms are fundamentally connected. In particular, we will show that all Nash equilibria (and in fact all equilibria that are based on local information only) are LVLs. Thus, if we assume that agents playing a game converge to an equilibrium, they are implicitly solving the corresponding computational task.

We note that the LOCAL model of distributed computing is particularly well-suited for realizing this connection. First, it is possible to prove *unconditional* impossibility results in the LOCAL model. Existing results cover many LVLs that are potentially interesting from the perspective of game theoretical applications (see e.g. [11, 35, 15, 6]). Second, any such impossibility results proven in the LOCAL model apply to a wide range of more realistic settings. In particular, they apply to game models where the play of the agents is constrained by the available information. This is due to the fact that algorithms in the LOCAL model are only limited by information propagation.

We show how to transfer existing impossibility results for computational tasks in the distributed setting to a model of games. To do this, we first establish an equivalence between the Nash equilibria of graphical games and locally verifiable labellings. Then, to transfer impossibility results to the model of fair sequential best responses, we show that the LOCAL model can simulate best responses. This implies that if all Nash equilibria of a game are hard to compute as LVLs, then the best-response dynamics cannot converge fast to these equilibria. Note that we study a notion of *families of graphical games* as an asymptotic generalisation of single games. These are defined formally in Appendix A.3. In what follows, when referring to a graphical game, we actually mean a family of graphical games.

Theorem 1 then states the foundational observation for our work: the Nash equilibria of graphical games can be seen as locally verifiable labelling problems.

▶ **Theorem 1** (Informal). *The Nash equilibria of a family of graphical games uniquely define an LVL.*

The LVL is constructed by taking all locally stable strategy assignments: Each node accepts a labelling if and only if it has no better response with respect to the utility function. We illustrate the correspondence in Figure 2.

Theorem 1 means that we can study LVLs to understand graphical games: We can, e.g., ask if all Nash equilibria of a game are hard to compute. If they are, this fact has implications for the underlying game. A formal version of the theorem is given in Appendix A.3.

3.1 Bounding convergence time of best responses

Next we discuss how impossibility results about LVLs can be turned into lower bounds for the convergence time of best responses. We show that best responses can be simulated in the LOCAL model. Formal versions of these results with their proofs are given in Appendix A.4.

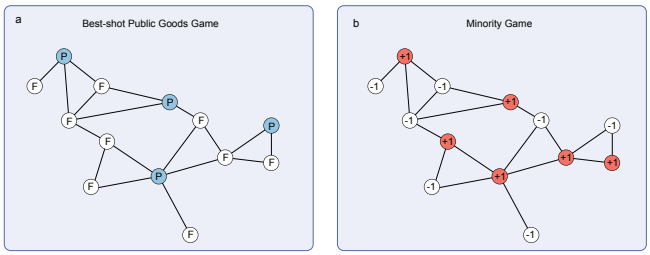


Figure 2 We illustrate the correspondence between Nash equilibria of graphical games and LVLs for the games described in Section 4. **a,** In the best-shot public goods game, nodes can pay a cost to produce a good (P) or to forego producing (F), relying on neighbors to produce. The Nash equilibria of this game are maximal independent sest of producers. **b,** The minority game gives players a choice between two resources. Their goal is to choose the opposite of the majority of their neighbors. The Nash equilibrium is a locally optimal cut: players have at least as many neighbors playing the opposite strategy as the same strategy.

▶ **Theorem 2** (Informal). *Best-response dynamics can be efficiently simulated in the LOCAL model.*

Proof sketch. The high-level idea is to compute an order of play such that many agents can move in parallel without affecting each other. Such an order is given by a suitable coloring of the underlying network. Each color class can move in parallel and the outcome is the same as if they had moved one by one. A coloring can be computed fast, it needs to be computed only once, and given a e.g. k-coloring one round of fair best responses can be simulated in k rounds in the LOCAL model. ◀

Since convergence time of best-response dynamics is defined as the worst case over all orders of play, Theorem 2 implies that if best-response dynamics converge fast, then there is a fast distributed algorithm as well. Conversely, if no fast algorithm exists, best-response dynamics cannot converge fast either.

▶ **Corollary 3** (Informal). *If the Nash equilibria of a family of games, as an LVL, have distributed complexity $\Omega(T)$, then best-response dynamics require $\Omega(T) - O(\log^* n)$ fair rounds to converge for that game.*

We want to highlight a particular implication of Corollary 3: together with the complexity gap of LVLs (Lemma 1) it implies a gap in the possible convergence times of best-response dynamics as well.

▶ **Corollary 4** (Informal). *Best-response dynamics of a graphical game either converge fast or take at least logarithmic time to converge.*

We finish this section by noting that in fact the applicability of impossibility results from distributed computing is much more general. Consider the following model of *local* dynamics: agents play one at a time, and when it is an agent's turn to move it decides its strategy based on all information inside its constant-radius neighborhood, for an arbitrarily large constant. There are no restrictions on the algorithm the agent uses to compute its decision, except that it is a function of the information available to it.

4 Classification of graphical games

In this section we describe a classification of families of graphical games based on Corollary 4, and then study examples of graphical games from three different classes.

We use two dimensions for this classification. The first dimension is computational hardness. Consider a graphical game family: computing a Nash equilibrium is either an easy or a hard problem. The second dimension is the convergence time of best-response dynamics: they also either converge fast or slow (at least logarithmic time). From this, we get the following types of game families:

1. There is a graphical game family such that a Nash equilibrium can be computed fast and best-response dynamics converge in constant number of rounds.
2. There is a graphical game family such that all Nash equilibria are hard to compute and therefore best-response dynamics converge slow.
3. There is a graphical game family such that a Nash equilibrium can be computed fast but best response dynamics converge slowly.

However, note that one combination is not possible: if computing a Nash equilibrium is hard, then best-response dynamics cannot converge fast by Corollary 3: distributed algorithms are at least as powerful as best-response dynamics.

4.1 Best-shot public goods game

This game family deals with the provision of public goods such as finding a cure for a disease or filling an important supply, assuming that only the maximal contribution counts towards the provision level instead of the sum of all players' contributions [18, 9]. In contrast to public goods games often studied as social dilemmas, i.e. where the social optimum is reached by all agents cooperating (producing) despite the incentive to do nothing, the problem here is not only one of free-riding, but also one of coordination. Agent groups have to figure out which of them should optimally be the one to provide the good, in order to avoid redundant costs. Here we consider the simplest version of the game where agents only have two choices, to provide a good, or not to provide it. More formally, each agent has two possible actions, i.e. $A_i = \{P, F\}$. Agents' utilities u_i are as following: if a focal agent plays F and one of their neighbors plays P, the utility is $u_i = 1$. If the agent plays P, $u_i = 1 - c$ (for some $0 < c < 1$), but if they and all their neighbors play F, the utility is 0. Simply put, providing the public good is costly, and it is preferable for an agent to have a neighbor do so; however, they are still better off providing it themselves than if nobody in their neighborhood does so.

The correspondence of this game family with distributed graph problems is a prominent one: the Nash equilibria of the best-shot public goods game correspond to *maximal independent sets* of agents playing P [9]. The set is independent (i.e. no two agents with P are adjacent), as two adjacent agents with the strategy P would have incentive to choose F. On the other hand the set is maximal (i.e. each agent plays P or has a neighbor that plays P) as otherwise such an agent would have incentive to play P.

Maximal independent set is an "easy" LVL [38]. Correspondingly, best responses converge in two fair rounds for the best-shot public goods game. We can compare this with the complexity analysis of best-shot public goods games by [32], and point out that our approach considers a natural concept of *distributed* complexity whereas previous work usually takes a different, more centralized view.

▶ **Theorem 5.** *Fair best responses in the best-shot public goods game converge in two rounds from any initial configuration.*

Proof. Assume that the system starts with some arbitrary strategy profile. We claim that after the first round the set of agents playing P is independent and after the second round it is maximal.

Assume that after the first round neighboring agents u and v play P. Then the one that played last would have seen that the other plays P, and their best response is F. Therefore the set of agents playing P is independent. In the second round no agent will switch from P to F, as all of their neighbors play F. If an agent plays F and their neighbors play F it will switch to P. After two rounds the agents playing P form a maximal independent set, which is a Nash equilibrium. ◀

The best-shot public goods game falls into the first category of our classification: Nash equilibria are easy to compute and best-response dynamics converge fast.

4.2 Minority game

Minority game is a basic anti-coordination and potential game (also called social game) [13, 8] where agents attempt to do the opposite of what their neighbors are doing. That is, they attempt to anti-coordinate with what the majority of their surrounding co-players do. This can be seen e.g. as a model competition for limited resources.

Agents have two possible actions, i.e. $A_i = \{-1, 1\}$. A focal player's utility u_i is defined as $u_i(a) = 1 + |\{j \in B(i) \setminus \{i\} \colon a_j \neq a_i\}| - |\{j \in B(i) \setminus \{i\} : a_j = a_i\}|$, i.e. the difference between the number of neighbors with a different label and the same label, plus 1 (for technical reasons to avoid Nash equilibria with utility 0).

The Nash equilibria correspond to strategy profiles where each agent has at least as many neighbors playing the opposite strategy as the same strategy. In this game, we again have a correspondence with a prominent graph problem: in distributed computing, the corresponding LVL is known as locally optimal cut. A *cut* is a binary labelling of the nodes, where an edge is said to be cut if its endpoints have different labels. A maximum cut maximises the number of cut edges, and a cut is locally optimal if no node can flip its label to improve the cut. Locally optimal cut is known to be a hard problem [6]. This immediately implies that best-response dynamics must also converge slow in minority games (Corollary 3). Minority games therefore fall into the second category of our classification: computing a Nash equilibrium is hard and best-response dynamics converge slowly.

4.3 Coloring game

Coloring games [30] are another family of basic anti-coordination games: each agent must choose a color from some palette of size k and it gains utility if and only if none of its neighbours choose the same color. Coloring games can be used to model scheduling and other resource allocation scenarios.

For a positive integer k, define the family of k-coloring games as follows: the action set is $A_i = \{1, 2, \ldots, k\}$ for each agent i. The utility function is $u_i(a) = 1$ if $a_i \neq a_j$ for all neighbours j of i, and 0 otherwise. That is, an agent receives utility 1 if it is properly colored, and 0 otherwise.

An important threshold for coloring games happens with $k = \Delta$ or $k = \Delta + 1$, where Δ is the maximum degree of the underlying network. If $k \geq \Delta + 1$, a k-coloring can be computed fast [7] and best responses also converge in one fair round. To see the latter, consider an arbitrary player making its move: since it has at most Δ neighbors, these neighbors can use at most Δ colors and it can always find a *free color* that gives it utility 1. After each agent has played once, everyone has chosen a free color, and this is a Nash equilibrium.

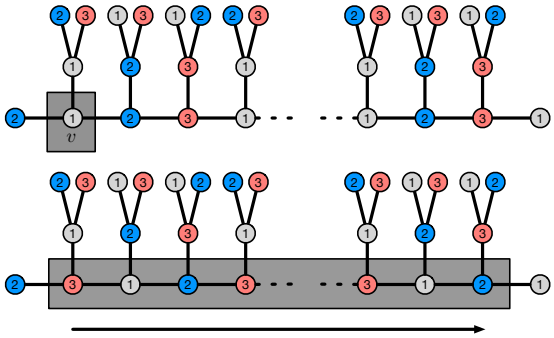


Figure 3 A lower bound construction for the convergence time of best-response dynamics for the Δ-coloring game. Top: initial configuration. Bottom: final stable configuration. Initially only the highlighted agent v is unstable: it has 3 as a free color. When it changes its strategy, it causes a cascade of agents, highlighted on the bottom, to change their strategies.

In contrast, finding a Δ-coloring is hard [11, 16]. This, however, does not necessarily mean that k-coloring games cannot converge fast. A greedy solution is no longer guaranteed, as the Δ neighbors of an agent could use all Δ colors. Thus the Nash equilibria of the coloring game do not match one to one with proper colorings. There exist Nash equilibria that are not proper colorings. This observation has an important implication, as there is a simple fast algorithm for finding a Nash equilibrium of any Δ-coloring game.

▶ **Theorem 6.** *There is fast algorithm that computes a Nash equilibrium of the Δ-coloring game, and the total utility of the Nash equilibrium is within factor $(1-\varepsilon)$ of the optimum.*

We describe and analyse the algorithm in the full version of the paper. In contrast, best-response dynamics may take up to linear time to converge:

▶ **Theorem 7.** *When $k \leq \Delta$, the worst-case convergence time of best responses for the k-coloring game is $\Omega(n)$.*

The idea for the proof is illustrated in Figure 3. The full proof is given in the full version of the paper.

This example shows that best-response dynamics can be computationally much less powerful than even simple distributed algorithms. In the next section we will show how distributed algorithms can be used as a tool to incentivise coordination and fast convergence for the coloring game.

5 Simulation games: incentivizing algorithmic cooperation

In this section, we introduce a novel way of incentivizing desired behavior in best-response dynamics. We show how to construct *simulation games*, where the best responses simulate a distributed algorithm. The Nash equilibria of simulation games then correspond to the outputs of the algorithm.

We illustrate two potential use cases for such games:

1. If a game has efficiently computable Nash equilibria, but best-response dynamics fail to converge, simulation games can be used to ensure convergence.
2. When best-response dynamics may converge to undesirable equilibria, simulation games can be used to select the Nash equilibria computed by an efficient distributed algorithm.

To incentivise the agents we assume they are guaranteed a certain utility if they follow the simulation game. This is the cost of the game, and do not try to optimise it. It is an interesting open question when simulation games could be made *self-sufficient*: the cost of incentive payments would be less than the utility gained from computing a better outcome. We present a case study in Section 5.2.

Simulation games always converge fast, requiring just one fair round. The drawback is that each agent needs to know the structure of the network and the strategies of its neighbours up to some constant distance, requiring additional means of communication.

The strategic guarantee of simulation games is that at each step, the best response of each agent is to continue to run the simulation, making these games strategy-proof. We note that even though there are multiple equilibria, we define them such that they all have the same value for a single agent. Therefore at no point should the agent take any action but their best response. Now if e.g. the agents would also get the utility from the outcome computed by the simulation game, this would no longer hold: there can be multiple equilibria, and some of these might be better than others. Agents could potentially have incentive to do something else than best response in order to affect the actions of the other agents.

5.1 Simulation games: overview

Simulation games are always based on the existence of a fast distributed algorithm. The algorithm defines the action sets and utility functions of the agents. Given an underlying network, these define a game where best responses converge to a simulation of the algorithm.

▶ **Theorem 8** (Informal). *Assume a fast distributed algorithm solves an LVL P for family of networks $\mathcal{N}$. Then for each network there exists a simulation game where best-responses converge in one round and Nash equilibria correspond to solutions computed by the algorithm.*

Simulation games are based on a special property of fast distributed algorithms: they can always be decomposed into a normal form with two parts [16]. In the first part, the input network is colored. In the second part, a specific version of the algorithm is applied with the coloring as input. The role of the coloring is to insulate the algorithm from the size of the input: we can use the same, constant-time algorithm on all networks. This makes it possible to encode the algorithm into a game.

5.1.1 Construction of simulation games

On a high level, simulation games are constructed as follows. We are given a network N and a fast distributed algorithm $\mathcal{A}$, with some constant running time T, as input. While T is a constant independent of the size of N, it can be larger than 1, the distance to which we assume agents in best-response dynamics "see". To simulate a distributed algorithm, the agents must receive information from some larger distance that is linear in T. This requires additional communication. To be consistent with the definition of graphical games, we model this communication by adding edges to N: the simulation game G is played on a network N' that is obtained by connecting all nodes in N within some specific constant distance $R = O(T)$. In addition, each agent initially knows the structure of its R-neighborhood in N.

The action sets are defined in two parts. The first part corresponds to all possible colorings of the neighbors of the agent. These are used to encode the first part in the normal form of the fast algorithm. The second part corresponds to the possible outputs of the algorithm. In addition there is an empty action, and we assume all agents start with this as their strategy.

The utility function is then used to incentivise correct simulations. For the first part, agents receive utility only if they choose a coloring of their R-neighborhood that is (1) a proper coloring (all agents have different color, a technical requirement) and (2) the coloring is consistent with the colorings chosen by other agents – all neighbors of a specific agent v choose the same color for v, and v agrees. The first part ensures that there is a properly defined notion of coloring for the T-neighborhood of each agent. For the second part, agents receive utility if they choose the output label that the algorithm $\mathcal{A}$ would choose with the coloring as input. This ensures that the agents choose an outcome $\mathcal{A}$ could have computed.

From this description we can see how the strategies will encode the output of $\mathcal{A}$. In particular, if $\mathcal{A}$ computes a Nash equilibrium of some game, then we can also directly project the Nash equilibria of the simulation game into Nash equilibria of the original game by considering only the output label.

Proof of Theorem 8, sketch. The most important properties of simulation games are that (1) they converge to Nash equilibria that correspond to outputs of the algorithm, and (2) they converge fast.

Initially all agents start with the empty strategy. When it is an agents time to move, it always gains utility by choosing an action that corresponds to a proper coloring of its neighbors. We ensure that there are enough colors so that this is always possible. Additionally, since the initial state is chosen to be empty, there are no conflicts with respect to the colors and agents always prefer to choose a coloring that is consistent with the colorings already chosen by other agents.

This ensures that in one fair round agents have chosen a proper coloring of the network. Given the coloring, the output labels are already defined: the agents only gain utility if they choose the output chosen by $\mathcal{A}$ given the input coloring. This ensures that the Nash equilibrium of the simulation game encodes a possible execution of the algorithm $\mathcal{A}$. Once the agents have reached one simulation, they have no incentive to change: all equilibria yield the same utility. Choosing the output label does not require additional communication: it can be chosen based on the coloring chosen by the agent. Therefore the best responses converge in one round. ◀

A formal description of simulation games and their construction is given in Appendix B.

5.2 Case study: coloring game

In this section we show how simulation games can be used to both ensure convergence and select equilibria in coloring games (see Section 4.3). Formal proofs are given in the full version.

First, consider the Δ-coloring game on general networks of maximum degree Δ. As shown in Section 4.3, this is a game where best-response dynamics require $\Omega(n)$ fair rounds to converge (Theorem 7). There is, however, an algorithm for efficiently computing a Nash equilibrium where $(1-\varepsilon)$-fraction of optimum total welfare (Theorem 6).

This shows that by constructing the simulation game of the algorithm from Theorem 6, we guarantee both fast convergence and almost optimal total utility.

▶ **Theorem 9** (Informal). *There is a simulation game that computes a Nash equilibrium of the Δ-coloring game with utility $(1-\varepsilon)$-fraction from the optimum.*

If we restrict the graph class to d-dimensional toruses, we can do even better. By previous work, it is known that for any constant d, there exists a fast algorithm for 4-coloring of d-dimensional toruses [12].

Consider the coloring game on two-dimensional n-by-n torus networks. That is, the set of nodes consists of $v_{i,j}\colon i,j \in \{0,1,\ldots,n-1\}$, there is an edge between nodes $v_{i,j}$ and $v_{i,k}$ if $k = (j+1) \bmod n$, and there is an edge between nodes $v_{i,j}$ and $v_{k,j}$ if $k = (i+1) \bmod n$. The complexity of k-coloring is completely understood in this setting:

▶ **Theorem 10** ([12]). *Computing a k-coloring on d-dimensional torus is easy when $k \geq 4$ and hard when $k = 2$ or $k = 3$.*

Both the 2-coloring problem and the 3-coloring problem are global (i.e. require $\Omega(n)$ rounds to compute). By Theorem 16 this implies that best responses also require at least $\Omega(n)$ rounds to converge. Even when best-response dynamics converge, they may converge to a Nash equilibrium where some agents are not properly colored.

In contrast, by Theorems 10 and 8, when $k \geq 4$ for each d-dimensional torus there exists a simulation game such that best-response dynamics converge in one round to an optimal solution: all agents are properly colored.

▶ **Theorem 11** (Informal). *There is a simulation game that computes a Nash equilibrium with optimal total utility for the 4-coloring game on any d-dimensional torus.*

Theorems 9 and 11 demonstrate how simulation games can be applied to games that have efficiently computable Nash equilibria but best-response dynamics do not converge efficiently.

6 Inefficiency of best responses

In this section we introduce and study a distributed analogue of Price of Anarchy [33]: what is the price of greedy behavior compared to optimal *distributed* coordination? The classic price of anarchy compares the total welfare of the optimal assignment to the worst Nash equilibrium. In a distributed setting, in particular, both of these are typically hard to compute. We want to replace these measures with more reasonable distributed analogues: what can be computed efficiently with locality T? In this way, our notion can be viewed as a refinement of the classic PoA concept from a distributed perspective.

Importantly, as we have seen in Section 3, efficient convergence is not possible in all graphical games. This means that an analogue of worst Nash equilibrium does not necessarily even exist! We consider a specific model of play, best-response dynamics, and compare the evolution of the state over time with respect to the best solution that can be computed with the same complexity. This leads to the notion of *inefficiency of best responses (IoBR)*.

This notion of inefficiency is computational inefficiency with respect to a time bound T. As a computational notion, it stands in contrast to the Price of Anarchy, which is an existential notion. Hence, it must be defined over families of games: a single game can always be solved perfectly by an algorithm that specifically encodes a solution to that game.

We show that there exist games such that we can bound the inefficiency of best responses away from the Price of Anarchy. This illustrates that the Price of Anarchy does not always fairly reflect the quality of solutions computed by best responses when time constraints are taken into account.

6.1 Defining inefficiency of best responses

To measure the performance of best responses, we compare the total welfare of a solution produced by T fair rounds of best responses to the best solution that can be computed in T rounds. The running time bound T is a function of the size of the game. We assume that the system starts from a random initial strategy profile. We consider random initial strategy

profiles, as worst-case strategy profiles make any computational arguments moot: the initial strategy profile could be the worst Nash equilibrium, even if it is hard to compute. To have a fair comparison, we consider what can be computed by randomised distributed algorithms.

Consider a family of games $\mathcal{G}$ and a distributed algorithm F with running time $T = T(n)$. For each $G \in \mathcal{G}$, let $F(G)$ denote total utility of the solution computed by F on G. Let $\mathrm{BR}(G, T)$ denote total utility of the solution produced in G by T rounds of best responses.

We define the T-inefficiency of best responses as

$$\mathrm{IoBR}(\mathcal{G}, T) = \max_{F \in \mathcal{F}_T} \min_{G \in \mathcal{G}} \frac{F(G)}{E[\mathrm{BR}(G, T)]},$$

where $\mathcal{F}_T$ denotes the family of all distributed algorithms with running time at most T. The definition compares the worst relative performance (over games of $\mathcal{G}$) of the best algorithm to the expected performance of best responses. The idea is that $\mathcal{G}$ represents some family of "similar" games: they are an asymptotic analogue of a single game in the definition of the Price of Anarchy.

To estimate the quantity $\mathrm{IoBR}(\mathcal{G}, T)$ we will bound all $F(G)$ from above using computational arguments, and bound $\mathrm{BR}(G, T)$ from below using both arguments about the behavior of best-response dynamics *and* computational arguments.

Note that when we consider the best strategy profile that is computable in T communication rounds, we consider distributed algorithms for *optimization* problems. That is, the algorithms might not compute a solution corresponding to some Nash equilibrium, but more generally any strategy assignment that tries to optimize the total welfare of all agents.

In the next section we show how the inefficiency of best responses can be estimated using tools from distributed computing.

6.2 Case study: best-shot public goods game

We begin by analysing the inefficiency of best responses in the public goods game. We show that the Price of Anarchy can be bounded away from the inefficiency of best responses. We prove that there exists an infinite family of best-shot public goods games such that even though good and bad solutions exist, no distributed algorithm can compute them efficiently. Therefore best responses cannot produce the bad solutions either. The following theorem states the outcome of our analysis for the best-shot public goods game.

▶ **Theorem 12** (Informal). *There exists a family $\mathcal{G}$ of public goods games such that for all $G \in \mathcal{G}$:*

1. *Price of Anarchy is the same for all $G \in \mathcal{G}$, and*
2. *Inefficiency of best responses, compared to fast algorithms, for $\mathcal{G}$ is strictly less than Price of Anarchy.*

The good solutions correspond to small *dominating sets* of producing agents: each agent has exactly one adjacent producer. The bad solutions are very redundant dominating sets: each non-producing agent has only producing neighbors.

To construct these games, we argue that there exist two families of networks:

1. In the first family, networks have both good and bad solutions, as described above.
2. In the second family, the networks are locally indistinguishable from networks of the first family, but do not have the bad and good solutions.

No algorithm can find the good or bad solutions in the second family, as they don't exist. Fast algorithms cannot distinguish between the two families. This means that fast algorithms cannot produce the good or bad solutions in the first family: if such algorithms existed, they would imply good and bad solutions for the second family as well, a contradiction.

We give a formal version of Theorem 12 and its proof in in the full version of this paper. There we also study the minority game and show a similar separation result for it: there exists a family of games such that Price of Anarchy is bounded away from the inefficiency of best responses.

7 Conclusion

In this work we introduced a novel approach to classifying the complexity of Nash equilibria in graphical games, and to understanding the convergence behavior of best-response and even general local dynamics. By establishing a connection to the analysis of distributed graph problems, we showed that the Nash equilibria of graphical games correspond to locally verifiable labelings: solutions to graph problems which are verifiable with constant round algorithms. Impossibility and complexity results provably transfer from the distributed setting to the game setting. In particular, the complexity gap observed in LVL problems also directly translates to the classification of equilibrium convergence times.

Thus, we can leverage this very natural connection to give lower bounds for convergence of best-response dynamics, to quantify the time-constrained inefficiency of best responses when convergence is slow or even absent, and to present how these results can be used for implementing mechanisms where best responses converge to a Nash equilibrium that is a solution of the corresponding graph problem. We also want to highlight once more that while our lower bounds are only strengthened by the simplicity of our examples, our results are relevant beyond the limited instances we have presented.

Our findings also relate to the open question of strategy proof algorithms for reaching equilibria in graphical games, as posed by Kearns in his 2007 survey. We note that in contrast to algorithms as in [29], our discussion indeed pertains to how agents *reach* equilibria while playing the game in a way that is rational with respect to their locally restricted knowledge. This poses an interesting avenue for further research.

In this work, we have only considered pure Nash equilibria. However, we note that our approach is much more general than that, since it first and foremost depends on information being locally restricted without making additional assumptions like the game being a potential game. One can prove similar results as in this work for *mixed Nash equilibria*, but even for different equilibrium concepts altogether, such as *correlated equilibria*. The latter correspond to a model where information is local, but can in some sense be exchanged between neighboring nodes, introducing correlated strategy distributions. A further simple extension of our model could see different local strategy update dynamics (e.g. ficitious play) at work, instead of restricting the analysis to best response dynamics only. This also highlights the connection of this research direction with evolutionary graph theory, which as a generalized approach to evolutionary dynamics features players with similarly bounded rationality [34]. Furthermore, it is also easily conceivable to analyze a far wider range of graphical games or LVLs with our approach, and to even extend the analysis to infinite graphs.

Naturally, there are limitations to our approach. Our techniques are relevant only in relatively sparse networks. Graphs of low diameter do not give impossibility results based on information propagation, as information can spread quickly. The results for such a setting would look quite different; furthermore, computation in a setting that comes closer to a centralized one where nodes do not have as strongly limited information is not well understood. Another issue lies with the fact that we only show lower bounds for convergence. This means that for instances that are not efficiently solvable, it could be the case that true convergence is far slower than what our results give. For further understanding, a complexity theory of the convergence of best-response dynamics is needed.

What we have considered in this paper should only represent a first taste for how powerful the connection between game theory and distributed computing is. We also highlight at this point that interpreting agents interacting in games on networks as a distributed system is highly intuitive. There are many possibilities to harness this, and we have only explored a very limited number of them. We hope that this work can serve as a proof of concept and be the starting point of exciting further research.

References

1 Guillermo Abramson and Marcelo Kuperman. Social games in a social network. *Physical Review E*, 63(3):030901, 2001. doi:10.1103/PhysRevE.63.030901.

2 James Aspnes, Kevin Chang, and Aleksandr Yampolskiy. Inoculation strategies for victims of viruses and the sum-of-squares partition problem. *Journal of Computer and System Sciences*, 72(6):1077–1093, 2006. doi:10.1016/J.JCSS.2006.02.003.

3 Guy Avni, Thomas A. Henzinger, and Orna Kupferman. Dynamic resource allocation games. In *Algorithmic Game Theory*, pages 153–166. Springer Berlin Heidelberg, 2016. doi:10.1007/978-3-662-53354-3_13.

4 Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems. *CoRR*, abs/1911.13294, 2019. arXiv:1911.13294.

5 Alkida Balliu, Juho Hirvonen, Janne H Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC 2018)*, pages 1307–1318. ACM Press, 2018. doi:10.1145/3188745.3188860.

6 Alkida Balliu, Juho Hirvonen, Christoph Lenzen, Dennis Olivetti, and Jukka Suomela. Locality of not-so-weak coloring. In *Proc. 26th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2019)*, volume 11639 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2019. doi:10.1007/978-3-030-24922-9_3.

7 Leonid Barenboim. Deterministic $(\Delta + 1)$-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *J. ACM*, 63(5):47:1–47:22, 2016. doi:10.1145/2979675.

8 Yann Bramoullé. Anti-coordination and social interactions. *Games and Economic Behavior*, 58(1):30–49, 2007. doi:10.1016/J.GEB.2005.12.006.

9 Yann Bramoullé and Rachel Kranton. Public goods in networks. *Journal of Economic Theory*, 135(1):478–494, 2007. doi:10.1016/J.JET.2006.06.006.

10 Sebastian Brandt. An automatic speedup theorem for distributed problems, 2019. arXiv:1902.09958.

11 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 Press, 2016. doi:10.1145/2897518.2897570.

12 Sebastian Brandt, Juho Hirvonen, Janne H Korhonen, Tuomo Lempiäinen, Patric R J Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pages 101–110. ACM Press, 2017. doi:10.1145/3087801.3087833.

13 Damien Challet and Y-C Zhang. Emergence of cooperation and organization in an evolutionary game. *Physica A: Statistical Mechanics and its Applications*, 246(3-4):407–418, 1997. doi:10.1016/S0378-4371(97)00419-6.

14 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. The Complexity of Distributed Edge Coloring with Small Palettes. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, pages 2633–2652. Society for Industrial and Applied Mathematics, 2018. doi:10.1137/1.9781611975031.168.

15 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model. In *Proc. 57th IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 615–624. IEEE, 2016. doi:10.1109/FOCS.2016.72.

16 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. *SIAM J. Comput.*, 48(1):122–143, 2019. doi:10.1137/17M1117537.

17 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.

18 Todd L Cherry, Stephen J Cotten, and Stephan Kroll. Heterogeneity, coordination and the provision of best-shot public goods. *Experimental Economics*, 16(4):497–510, 2013. doi:10.1007/s10683-012-9349-1.

19 Simon Collet, Pierre Fraigniaud, and Paolo Penna. Equilibria of games in networks for local tasks. In *Proc. 22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *LIPIcs*, pages 6:1–6:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.OPODIS.2018.6.

20 Constantinos Daskalakis and Christos H Papadimitriou. Computing pure nash equilibria in graphical games via markov random fields. In *Proceedings of the 7th ACM Conference on Electronic Commerce*, pages 91–99, 2006. doi:10.1145/1134707.1134718.

21 Constantinos Daskalakis and Christos H. Papadimitriou. On a network generalization of the minmax theorem. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming*, pages 423–434, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-02930-1_35.

22 Edith Elkind, Leslie Ann Goldberg, and Paul Goldberg. Nash equilibria in graphical games on trees revisited. In *Proc. 7th ACM Conference on Electronic Commerce*, pages 100–109. Association for Computing Machinery, 2006. doi:10.1145/1134707.1134719.

23 Patrik Floréen, Petteri Kaski, Valentin Polishchuk, and Jukka Suomela. Almost stable matchings by truncating the gale-shapley algorithm. *Algorithmica*, 58(1):102–118, 2010. doi:10.1007/S00453-009-9353-9.

24 David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962. doi:10.2307/2312726.

25 Mohsen Ghaffari and Hsin-Hao Su. Distributed Degree Splitting, Edge Coloring, and Orientations. In *Proc. 28th ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 2505–2523. Society for Industrial and Applied Mathematics, 2017. doi:10.1137/1.9781611974782.166.

26 Samuel Ieong, Robert McGrew, Eugene Nudelman, Yoav Shoham, and Qixiang Sun. Fast and compact: A simple class of congestion games. In *Proc. 20th National Conference on Artificial Intelligence – Volume 2 (AAAI 2005)*, pages 489–494. AAAI Press, 2005. URL: http://www.aaai.org/Library/AAAI/2005/aaai05-077.php.

27 Matthew O Jackson and Leeat Yariv. Diffusion of behavior and equilibrium properties in network games. *American Economic Review*, 97(2):92–98, 2007. doi:10.1257/aer.97.2.92.

28 Matthew O Jackson and Yves Zenou. Games on networks. In *Handbook of game theory with economic applications*, volume 4, pages 95–163. Elsevier, 2015. doi:10.1016/B978-0-444-53766-9.00003-3.

29 Michael Kearns. Graphical games. *Algorithmic game theory*, 3:159–180, 2007. doi:10.1017/CBO9780511800481.009.

30 Michael Kearns, Siddharth Suri, and Nick Montfort. An experimental study of the coloring problem on human subject networks. *Science*, 313(5788):824–827, 2006. doi:10.1126/science.1127207.

31 Michael J. Kearns, Michael L. Littman, and Satinder P. Singh. Graphical models for game theory. In *Proc. 17th Conference in Uncertainty in Artificial Intelligence (UAI 2001)*, pages 253–260, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=107&proceeding_id=17.

32 Zohar Komarovsky, Vadim Levit, Tal Grinshpoun, and Amnon Meisels. Efficient equilibria in a public goods game. In *Proceedings of the 2015 IEEE / WIC / ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT) – Volume 01*, WI-IAT '15, pages 214–219. IEEE Computer Society, 2015. doi:10.1109/WI-IAT.2015.91.

33 Elias Koutsoupias and Christos H. Papadimitriou. Worst-case equilibria. In *Proc. 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1999)*, volume 1563 of *Lecture Notes in Computer Science*, pages 404–413. Springer, 1999. doi:10.1007/3-540-49116-3_38.

34 E. Lieberman, C. Hauert, and M. A. Nowak. Evolutionary dynamics on graphs. *Nature*, 433:312–316, 2005. doi:10.1038/nature03204.

35 Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

36 Moni Naor and Larry Stockmeyer. What Can be Computed Locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

37 Luis E Ortiz and Michael Kearns. Nash propagation for loopy graphical games. *Advances in neural information processing systems*, pages 817–824, 2003. doi:10.5555/2968618.2968720.

38 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001. doi:10.1007/PL00008932.

39 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

40 Tim Roughgarden. Routing games. *Algorithmic game theory*, 18:459–484, 2007.

41 Grant R Schoenebeck and Salil Vadhan. The computational complexity of nash equilibria in concisely represented games. *ACM Transactions on Computation Theory (TOCT)*, 4(2):1–50, 2012. doi:10.1145/2189778.2189779.

42 Richard Southwell, Yanjiao Chen, Jianwei Huang, and Qian Zhang. Convergence dynamics of graphical congestion games. In Vikram Krishnamurthy, Qing Zhao, Minyi Huang, and Yonggang Wen, editors, *Game Theory for Networks*, pages 31–46. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-35582-0_3.

43 Alexander J Stewart, Mohsen Mosleh, Marina Diakonova, Antonio A Arechar, David G Rand, and Joshua B Plotkin. Information gerrymandering and undemocratic decisions. *Nature*, 573(7772):117–121, 2019. doi:10.1038/S41586-019-1507-6.

44 David Vickrey and Daphne Koller. Multi-agent algorithms for solving graphical games. *AAAI/IAAI*, 2:345–351, 2002. URL: http://www.aaai.org/Library/AAAI/2002/aaai02-053.php.

A Preliminaries

A.1 Defininig LVLs

Formally LVLs are defined as follows.

▶ **Definition 13** (Locally verifiable labelings). *An LVL P is a computational problem parametrized by a constant maximum degree Δ, and consists of an input alphabet Σ, an output alphabet Γ, and a set of configurations $\mathcal{C}$. The input alphabet Σ is finite set of labels, and the output alphabet Γ is possibly an infinite set (this generalises LCLs). Each configuration $C \in \mathcal{C}$ is a graph G centered on some node v with radius at most k for some constant k (the verification radius of P). Each node of C is labelled with some element $\sigma \in \Sigma$ and $\gamma \in \Gamma$.*

An instance of P is a pair (N, x) where $N = (V, E)$ is a network of maximum degree Δ and x is a mapping $x\colon V \to \Sigma$. If P has no input labels (i.e. $|\Sigma| = 1$), any network N is an instance of P. We say that a mapping $f\colon V \to \Gamma$ is a solution to P on (N, x) if and only if each k-neighborhood of N labelled with x and f is a configuration in $\mathcal{C}$.

We note here that in this work, we only consider LVLs with radius $k = 1$, as this matches the definition of graphical games. The definition assumes a constant bound on the maximum degree of the input network. Throughout this work, we will assume that networks have a constant maximum degree Δ. This is important assumption from the perspective of distributed complexity theory, as it is very different in dense networks (large maximum degree).

A.2 Complexity gap of LVLs

There is a gap in the possible complexities of LVL problems: they can either be solved very fast or require logarithmic time.

▶ **Fact 2** (Chang et al. [16]). *On general bounded-degree graphs, i.e., if the maximum degree of the graph N is bounded by a constant Δ, the deterministic distributed complexity of an LVL is either $O(\log^* n)$, or $\Omega(\log n)$. The randomized complexity is either $O(\log^* n)$ or $\Omega(\log \log n)$.*

Here, $\log^* n$ is the *iterated logarithm* (pronounced "log-star"), a function which grows significantly more slowly than the logarithm: e.g. $\log^*$ of the number of atoms in the observable universe is 5. Formally, $\log^* n$ is defined as:

$$\forall x \leq 2 : \log^* x := 1, \quad \forall x > 2 : \log^* x := 1 + \log^*(\log x)$$

The complexity gap on bounded-degree networks is also the best possible: there exists an LVL such that its deterministic complexity is $\Theta(\log n)$ and the randomized complexity is $\Theta(\log \log n)$ [16, 11, 25]. This also proves that the deterministic and randomized complexities of an LVL can be *exponentially separated*.

On other graph families, the complexity gap between easy and hard problems can be even larger. For example, on paths and cycles LVLs have complexity either $O(\log^* n)$ or $\Theta(n)$ [16], and on two-dimensional grids and toruses either $O(\log^* n)$ or $\Omega(\sqrt{n})$ [12].

Throughout this paper, we will call a distributed algorithm with complexity $O(\log^* n)$ *fast*.

A.3 Correspondence of LVLs and Nash equilibria of graphical games

To study the asymptotic hardness of computing equilibria, we define *families* $\mathcal{G}$ of graphical games, parametrized by a constant degree bound Δ and some (possibly infinite) set $\mathcal{N}$ of networks of maximum degree Δ. For example, this could be all networks of maximum degree Δ, or all trees of maximum degree Δ.

▶ **Definition 14** (Family of graphical games). *A family of graphical games then consists of all possible games on $\mathcal{N}$ generated from some base set of rules. The* type *of an agent is a pair (A, u), where A is an action set and u is a utility function. Fix Δ finite collections of types $\mathcal{T} = (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\Delta)$, one for each possible degree of an agent. Then the family of games $\mathcal{G} = \mathcal{G}(\mathcal{N}, \Delta, \mathcal{T})$ is defined as the set of all possible games on each $N \in \mathcal{N}$ such that for each degree d, each agent of degree d is assigned an action set and utility function according to some type in $\mathcal{T}_d$.*

Using this definition we can construct a locally verifiable labelling that captures exactly the Nash equilibria of the family of graphical games.

▶ **Theorem 15.** *Let $\mathcal{G}$ be a family of graphical games. There is a locally verifiable labelling $P(\mathcal{G})$ such that for each $G \in \mathcal{G}$ the pure strategy Nash equilibria of G correspond exactly to solutions of P.*

Proof. This follows from the locality of the utility functions of graphical games: whether each agent is stable only depends on the strategies of its neighbours, and a strategy profile is a Nash equilibrium if and only if all agents are stable.

Consider an arbitrary family of graphical games $\mathcal{G}$. To construct the corresponding LVL P, do the following. The set of input labels is the union of the types of $\mathcal{G}$. The set of output labels Γ is the set of all possible pure strategies, i.e. the union of action sets of the types of $\mathcal{G}$. To construct the set of configurations C of P, let $\mathcal{B}$ denote all possible 1-neighbourhoods of $\mathcal{G}$ labelled with types, over all $G \in \mathcal{G}$. Consider an arbitrary such 1-neighbourhood $B \in \mathcal{B}$ centered at some agent i. Each agent $j \in B$ is labelled with a type (A_j, u_j). Consider all possible strategy profiles a drawn from the action sets of the agents. For each a, if a_i is a best response of i with respect to a_{-i}, add B with the input labelling given by the types and the output labelling given by a to C.

By construction, for each specific $G \in \mathcal{G}$ there exists a corresponding instance of P such that the solutions are exactly the Nash equilibria of G. ◀

It should be stressed that the correspondence does not use any properties specific to pure Nash equilibria. For example mixed strategy Nash equilibria of graphical games can also be used to define LVLs analogously: the output labels of the LVL consist of mixed strategies instead of pure strategies, as above.

A.4 Simulating best responses

In this section we show how best-response dynamics can be simulated in the LOCAL model with small overhead.

Let $G = (A, u, N)$ be a graphical game. We construct a corresponding instance in the LOCAL model by taking the network N and labelling each agent with its type. Then, if we consider the deterministic LOCAL model, an adversary assigns $O(\log n)$-bit names to the nodes. In the randomized LOCAL model, each node gets a uniformly random infinite string of bits as input instead.

We define the convergence time of best-response dynamics for a family of graphical games as the worst-case convergence time over all games. We show that if the best responses converge in $T(n)$ rounds, where n is the size of the underlying network, then this can be turned into a distributed algorithm that computes the corresponding Nash equilibrium in $O(\log^* n + T(n))$ rounds.

▶ **Theorem 16** (best responses correspondence)**.** *Fix a function T. Consider a family $\mathcal{G}$ of graphical games.*

1. *If the best-response dynamics converge on each $G \in \mathcal{G}$ in $T(n)$ rounds from a constant initial strategy profile, then there exists a deterministic distributed algorithm in the LOCAL model that solves the LVL $P(\mathcal{G})$ corresponding to the (pure) Nash equilibria of $\mathcal{G}$ on each G in $O(\log^* n + T(n))$ rounds.*
2. *If the best-response dynamics converge with high probability on each $G \in \mathcal{G}$ in $T(n)$ rounds from a random initial strategy profile, then there exists a randomized distributed algorithm in the LOCAL model that solves the LVL $P(\mathcal{G})$ corresponding to the (pure) Nash equilibria of $\mathcal{G}$ on each G in $O(\log^* n + T(n))$ rounds with high probability.*

Before proving the theorem we show a helper lemma that shows how best-response dynamics can be simulated from any specific initial state.

▶ **Lemma 17.** *Assume that $(\mathcal{A}, u, N)$ is a graphical game and a is some strategy profile. A distributed algorithm that is given a as an input can simulate T rounds of best responses, for some ordering of the play, in $O(\log^* n + T)$ rounds.*

Proof. The simulation consists of two phases. In the first phase, the nodes compute a coloring of N^2 (the virtual network obtained by connecting all nodes at distance at most 2 in N) with $k = \Delta^2 + 1$ colors. That is, each node v chooses a label $c(v)$ from $\{1, 2, \ldots, k\}$ such that any two nodes u and v within distance 2 in N have different labels $c(u) \neq c(v)$. This can be computed in $O(\log^* n)$ rounds [7].

In the second phase this coloring is treated as a *schedule*: at round j of the second phase each node with color $i = j \bmod k$ is active, applies the best response to the current strategy profile, and sends its new strategy to its neighbors. The key is that any two nodes updating their strategy at the same time do so *independently*: since they are not neighbors, their choices do not depend on each other. Therefore applying best responses at all nodes of the same color class is equivalent to letting the corresponding agents play in any sequential order: given an initial strategy profile a, all orderings produce the same strategy profile a'. Simulating all color classes one by one therefore corresponds to *some* ordering of sequential play.

Since there are $k = \Delta^2 + 1$ color classes, simulating one fair round of best responses takes k rounds in the LOCAL model. Since we assume Δ is a constant, simulating T rounds of best responses can be done in $O(T)$ rounds. With the initial coloring step we have that the total running time of the simulation is $O(\log^* n + T)$, as required. ◀

It follows that simulating best responses until convergence can be done with an additive $O(\log^* n)$ overhead.

Proof of Theorem 16. First, assume that the best responses start from a constant or worst-case initial strategy profile. Each node can simply choose the same initial value and simulate $T(n)$ rounds of best responses by Lemma 17. Since we assume that best responses converge for any order of play in $T(n)$ rounds, it follows that $T(n)$ rounds of simulation converge as well. Computing the simulation until convergence takes $O(\log^* n + T(n))$ rounds in the deterministic LOCAL model.

Next, assume that the best responses start from a random initial strategy profile. Now it is no longer possible to use deterministic algorithms to run the simulation. Using the random inputs, each node can choose a random initial strategy. Then it can simulate best responses for $T(n)$ rounds by Lemma 17. Since we assume that the best responses converge with high probability and the dynamics are deterministic given the initial configuration, the simulation also converges with high probability in $T(n)$ rounds. The simulation can be computed in $O(\log^* n + T(n))$ rounds in the randomized LOCAL model. ◀

As shown in Theorem 15, a family of graphical games $\mathcal{G}$ defines an LVL $P(\mathcal{G})$. It also defines a family of problem instances of P: each G can be seen as network N with input labelling given by the types of the agents.

▶ **Corollary 18.** *Assume that problem of solving the LVL $P(\mathcal{G})$ over the instances defined by $\mathcal{G}$ has deterministic complexity $\Omega(T(n))$ and randomized complexity $\Omega(T'(n))$, for any $T(n), T'(n) = \Omega(\log^* n)$. Then best-response dynamics for $\mathcal{G}$ require $\Omega(T(n))$ and $\Omega(T'(n))$ fair rounds to converge from a constant and a randomized initial strategy profile, respectively.*

Proof. This follows from Theorem 16: if best-response dynamics converge faster, then this can be turned into a fast distributed algorithm, a contradiction. ◀

The complexity gap of LVLs in the LOCAL model, in the context of our result, implies that if the Nash equilibria of a game are not efficiently computable, then best responses converge significantly slower.

▶ **Corollary 19.** *Let $\mathcal{G}$ be a family of games such that $P(\mathcal{G})$ cannot be solved in time $O(\log^* n)$. Then the best-response dynamics for $\mathcal{G}$ require $\Omega(\log n)$ fair rounds to converge from constant initial strategy profile, and $\Omega(\log \log n)$ fair rounds to converge from a random initial strategy profile.*

We emphasize that the deterministic and randomized convergence time lower bounds hold even if we consider best-response dynamics that start respectively from any constant or a randomized initial state.

B Simulation games

B.1 Simulation games: Overview

Simulation games are based on the existence of fast distributed algorithms: assume there exists an algorithm A with running time $T(n) = O(\log^* n)$ solving some LVL P. Then for any network N we can construct a graphical game G where best-response dynamics converge in one fair round to a Nash equilibrium that encodes the output of A on N. In particular, if A computes Nash equilibria for some family of games, then best-response dynamics in the corresponding simulation games will also converge to the Nash equilibria computed by A.

Our construction has several desirable properties.

1. Convergence time. Best responses converge in one fair round.
2. Equilibrium selection. Since best responses converge to an output of algorithm A, the resulting Nash equilibria inherit the properties of solutions computed by A. In Section 5.2 we show how this can be used to ensure convergence to a desirable subset of Nash equilibria.
3. Locality. It would be trivial to construct a new game with the above two properties: an outside observer could set a utility function structured such that agents have to choose a specific outcome. This, however, would essentially amount to a centralised authority assigning the strategies to the agents. In our construction, each agent only receives the description of the algorithm A and information about other agents up to some constant distance, independent of the size of the underlying network. This means that assuming limited local communication, the agents can set up this simulation game locally without centralised control.

Due to the locality of simulation no similar constructions exist for games that don't have efficiently solvable Nash equilibria: this would also imply efficient algorithms for the corresponding LVL problems.

On a high level, simulation games are constructed as follows: We are given a network N and an efficient algorithm $\mathcal{F}$. The actions of each agent consist of all possible correct local executions of the algorithm $\mathcal{F}$ on N. The utility of an agent is 1 if and only if its strategy is an execution that is compatible with the executions of its neighbors. As a correct execution always exists, best responses converge to such an execution when starting from an empty initial state (i.e. each agent has a special empty action).

The following theorem is the formal version of Theorem 8.

▶ **Theorem 20.** *Let $\mathcal{F}$ be a distributed algorithm that solves LVL P in time $O(\log^* n)$ in networks of maximum degree Δ. Then for each P-instance (N, x) there exists a simulation game $G = (A, u, N')$ with the following properties.*

(P1) *Best responses converge in one fair round from the empty initial strategy profile in G.*

(P2) *Nash equilibria of G correspond to legal outputs of $\mathcal{F}$.*

(P3) *Given (N, x) and $\mathcal{F}$, simulation game G can be constructed in $O(t)$ rounds with a distributed algorithm.*

A particularly interesting use case for the simulation games is when a fast algorithm computes the Nash equilibria of a graphical game.

▶ **Corollary 21.** *If $\mathcal{F}$ solves LVL $P(\mathcal{G})$ corresponding to the Nash equilibria of a family of graphical games $\mathcal{G}$, then the simulation game converges to a Nash equilibrium of $\mathcal{G}$.*

We will next give the technical details for the construction of simulation games.

B.2 Constructing simulation games

To define simulation games, we need to consider algorithms in a specific *normal form*, the existence of which is implied by the speedup result of [16]. Lemma 22 (stated in the following section) states that $O(\log^* n)$-time algorithms can be decomposed into two phases. In the first phase the algorithm computes a distance-$(2t+2)$ coloring for some constant parameter t that depends on the algorithm. Then a t-round algorithm is applied with the coloring taking the role of unique identifiers. We say that this is the t-normal form of the algorithm. In simulation games the best responses construct these colorings and then choose the output of the algorithm on that particular coloring.

Now let $\mathcal{F}$ be a distributed algorithm in t-normal form that solves LVL P. For a P instance (N, x) with maximum degree Δ, define the *simulation game of $\mathcal{F}$ on N* as $G = (A, u, N')$ as follows.

1. The set of agents are the nodes of N. Construct a new network N' by connecting two nodes u and v if and only if their distance in N is at most $4t + 2$ (some constant).
2. The actions A_v of each agent v encode the possible locally correct simulations of $\mathcal{F}$. This is defined in two parts R_v and L. The first set R_v consists of all possible labellings of the t-neighborhood of v in N with distinct colors from $\{1, 2, \ldots, \Delta^{2t+2} + 1\}$. The second part L consists of the possible output labels of P. Include the pair (r, ℓ) in A_v if and only if $\mathcal{F}$ would output ℓ at v given x as the input labelling and r as the input coloring of $B(v, t)$. In addition there is the empty action.
3. The utility functions u_v encode the correct simulations. The utility $u_v(s) = 1$ if and only if the following hold. First, the coloring $r_v \in R_v$ is *compatible* with the colorings $r_u \in R_u$ of each neighbor u with a non-empty strategy. That is, for each $w \in B_N(v, t) \cap B_N(u, t)$, we have that $r_v(w) = r_u(w)$ or r_u is empty. Second, the colorings form a proper $(2t + 2)$-hop coloring of N. That is, if we map all the compatible colorings r_v for all v onto N, then two nodes in N at distance at most $2t + 2$ have distinct colors. Since two agents are connected in N' if they are within distance $4t + 2$ in N, it is possible to encode this in u. The color assigned to each agent at distance t from some agent v must differ from the colors of other agents within distance $2t + 2$ of it. These agents can only be colored by agents within distance $4t + 2$. Note that by construction we only allow actions where ℓ corresponds to the correct output of $\mathcal{F}$ so there is no need to encode this in the utility function.

   For the empty action and for strategy profiles that do not have these properties the utility is 0.

We will show that simulation games converge in one fair round to an equilibrium that is equivalent to an equilibrium of the original game.

We also note that similar constructions do not exist for algorithm that solve hard problems. If they did, by Theorem 16 we would get a faster algorithm for solving the problem, a contradiction.

B.3 Proving the properties of simulation games

Before proving Theorem 20, we need the following technical lemma. It establishes that $O(\log^* n)$-time solvable LVLs can also be solved by algorithms in the required normal form.

▶ **Lemma 22.** *Assume that LVL P can be solved in $O(\log^* n)$ rounds by a deterministic distributed algorithm. Then there exists an algorithm $\mathcal{F}$ in the following normal form: the algorithm runs in $t = O(1)$ rounds (for some t dependent on P), it takes a $(2t+2)$-hop c-coloring, for $c = \Delta^{2t+2} + 1$, as an input, and outputs the solution to P.*

The proof follows from the speedup theorem of [16]. A similar normal form construction has been used by Brandt et al. [12].

We are now ready to prove the properties of simulation games.

Proof of Theorem 20. We show that the strategies of agents who have already played will constitute a correct partial simulation in N'. Initially this is trivially true, as all agents are assumed to start from the empty strategy.

Now assume that a is the strategy profile after some number of best responses such that the non-empty strategies agree on the color of each agent and assume that some agent v is scheduled to play. Since the utility is 0 if the agent chooses an action that is not compatible as a coloring, it must choose a compatible coloring. Since we assumed that a encodes a partial coloring and there are always enough colors to choose from (i.e. there are $\Delta^{2t+2} + 1$ colors), v can always choose an action that gives it utility 1.

After one fair round, each agent v has chosen a strategy such that the colorings r_v agree on all applicable nodes, and this is a Nash equilibrium of G'. The output label σ_v of each action, by definition, corresponds to the output of $\mathcal{F}$ on (N, x) given the input coloring that the agents have chosen.

Finally, it remains to show that G' can be constructed efficiently in the LOCAL model on the network N. This is achieved using a standard approach. First each node v gathers its $(4t+2)$-hop neighborhood and outputs its neighbors in N'. Since the algorithm has access to $\mathcal{F}$, the algorithm in the normal form, it can consider every coloring of $B(v, t)$ and form the action set A'_v. Finally, since the algorithm has access to the $(4t+2)$-neighborhood of v in (N, x), it can compute the value of the utility function u_v for all possible strategies of the neighbors. ◀

Eating Sandwiches: Modular and Lightweight Elimination of Transaction Reordering Attacks

Orestis Alpos
Institute of Computer Science, University of Bern, Switzerland

Ignacio Amores-Sesar
Institute of Computer Science, University of Bern, Switzerland

Christian Cachin
Institute of Computer Science, University of Bern, Switzerland

Michelle Yeo
IST Austria, Klosterneuburg, Austria

Abstract

Traditional blockchains grant the miner of a block full control not only over which transactions but also their order. This constitutes a major flaw discovered with the introduction of decentralized finance and allows miners to perform MEV attacks. In this paper, we address the issue of sandwich attacks by providing a construction that takes as input a blockchain protocol and outputs a new blockchain protocol with the same security but in which sandwich attacks are not profitable. Furthermore, our protocol is fully decentralized with no trusted third parties or heavy cryptography primitives and carries a linear increase in latency and minimum computation overhead.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Consensus, MEV, Byzantine behavior, Rational behavior

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.12

Related Version *Full Version*: `https://arxiv.org/abs/2307.02954`

Funding This work has been funded by the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021_188443 (Advanced Consensus Protocols).

Acknowledgements We would like to thank Krzysztof Pietrzak and Jovana Mićić for useful discussions.

1 Introduction

The field of blockchain protocols has proved to be extremely robust. Since its creation with Bitcoin [45], it had gone through several enhancements such as Ethereum [2] and has seen the appearance of *decentralized finance* (DeFi). With this, some design flaws started to show up. Blockchains would ideally allow users to trade tokens with each other in a secure manner. However, existing designs do not consider users trading tokens of one platform for FIAT currency or tokens of a different platform, arguably one of the major flaws of today's blockchain platforms, *maximal extractable value* (MEV) [26]. Current estimates show that the total volume of MEV since 2020 is around 675M USD [3]. From a social welfare perspective, while MEV is profitable to miners, it presents a serious invisible tax on the users on the blockchain. Indeed the financial losses built up over time could potentially shy away users from the blockchain, and consequently impact the security of the chain.

Sandwich attacks are one of the most common types of MEV [38] accounting for a loss of 174M USD over the span of 33 months [48] for users of Ethereum. Sandwich attacks leverage the miner's ability to *select* and *position* transactions within a block. Consider the simple example of a sequence of transactions that swap one asset X for another asset Y in a

decentralized exchange where exchange rates are computed automatically based on some function of the number of underlying assets in the pool (e.g., a constant product market maker [7]). Now suppose there is a miner that also wants to swap some units of X for Y. The most favorable position for the miner would be to place their transaction at the start of the sequence, so as to benefit from a lower X-to- Y exchange rate. This approach achieves a simple arbitrage strategy for any sequence of X-to-Y swaps: the miner can insert an X-to-Y exchange at the start of the sequence and use the computed exchange rate to sell, say, k units of X to get units of Y. The miner then front-runs the sequence of X-to-Y swaps, i.e., it inserts its own transaction at the start. To finish off the attack, the miner back-runs the sequence with another transaction of its own that swaps some units of Y to X, i.e., inserts this transaction at the end, and will often obtain more than k units of X. In this way, the miner profits from its insider knowledge and its power to order transactions. Refer to the full version [6] for a detailed description of exchange rate computation and sandwich attacks.

Since any miner of a given block has full control over the transactions added to the block, as well as over the way transactions are ordered, it is straightforward for the miner to launch the above attack. Consequently, this gives miners a lot of power as they control precisely the selection and positioning of transactions with every block they mine. This problem has received broad attention in the practice of DeFi and in the scientific literature.

A classic technique to mitigate this attack is thus to remove the control over the positioning of the transactions in the block from the adversary, whether by using a trusted third party to bundle and order the transactions as in flashbots[1], Eden[2], or OpenMEV[3]. Another method works by imposing a fair ordering of the transactions using a consensus algorithm that respects the order in which miners and validators first received the transactions [34]. The classical solutions make the protocol dependent of external factors, besides, affecting efficiency.

In this work, we introduce the *Partitioned and Permuted Protocol*, abbreviated Π^3, an efficient decentralized algorithm that does not rely on external resources to counter front-running. It renders sandwich attacks unprofitable and can easily be implemented on top of an existing blockchain protocol Π.

Protocol Π^3 determines the final order of transactions in a block B_i, created by a miner M_i, through a *uniformly randomly chosen permutation* Σ_i. To explain the method, let us focus on three transactions in B_i, a victim transaction tx^* submitted by a client, and the front-running and back-running transactions, tx_1 and tx_2, respectively, created by the miner. Since any relative ordering of these three transactions is equally probable, tx_1 will be ordered before tx_2 with the same probability as tx_2 before tx_1, hence the miner will profit or make a loss with the same probability. Protocol Π^3 uses a fresh permutation for each block; it is chosen by a set of *leaders*, which are recent miners in the blockchain. We recognize and overcome the following challenges.

First, Σ_i must not be known before creating B_i, otherwise M_i would have the option to use Σ_i^{-1}, the inverse of Σ_i, to initially order the transactions in B_i, so that the final order is the one that benefits M_i. We overcome this by making Σ_i known only *after M_i has been mined*. On the other hand, if Σ_i is chosen after creating B_i, a coalition of leaders would be able to try multiple different permutations and choose the most profitable one – the number of permutations a party can try is only limited by their processing power. For these reasons, we have the leaders *commit* to their contributions to Σ_i before B_i becomes known, producing

[1] `https://www.flashbots.net`
[2] `https://www.edennetwork.io`
[3] `https://openmev.xyz/`

unbiased randomness. To incentivize leaders to open their commitments, our protocol Π^3 employs a delayed reward release mechanism that only releases the payment to leaders when they have generated and opened all commitments.

In some cases, however, performing a sandwich attack might still be more profitable than the block reward, and hence a leader might still choose to not reveal their commitment to bias the resulting permutation. In general, a coalition of k leaders can choose among 2^k permutations out of the $n_t!$ possible ones, where n_t denotes the number of transactions in the block. It turns out that the probability that tx_1, tx^*, and tx_2 appear in that order in one of the 2^k permutations can be significant for realistic values. Protocol Π^3 mitigates this by *dividing each transaction* into m chunks, which lowers the probability of a profitable permutation in two ways. First, the number of possible permutations is much larger, $(n_t m)!$ instead of $n_t!$. Second, a permutation is now profitable if the majority of chunks of tx_1 appear before the chunks of tx^*, and vice versa for the chunks of tx_2. As we discuss, the probability of a profitable permutation approaches zero rapidly as the number of chunks m increases. We discuss how to implement the chunking mechanism while preserving transaction integrity and atomicity.

Organization. In this paper, we introduce a construction that takes as input a blockchain protocol Π and produces a new blockchain protocol Π^3 in which sandwich attacks are no longer profitable. We begin by revisiting the concept of *atomic broadcast* [17] and setting the model for the analysis. Secondly, we introduce our construction justifying how miners are incentivized to follow the protocol before moving on to analyzing the construction in detail. Thirdly, we guarantee that the construction does not include any vulnerability to the protocol by showing that Π^3 implements a variant of atomic broadcast if Π does. This part of the analysis is performed in the traditional *Byzantine* model. Fourthly, we consider the *rational* model to show that sandwich attacks are no longer profitable in Π^3. We consider the dual model of *Byzantine* for the security analysis and *rational* for the analysis of the sandwich attacks because we considered it to be a perfect fit to show that the security of Π^3 is not weakened even against an adversary that obtains nothing for breaking the protocol, as well as, we can assume that any party attempts to extract value from any sandwich attack. In other words, we consider both the security analysis and the analysis of the sandwich attack in the worst scenario possible for the protocol. Lastly, we conclude the paper with an empirical analysis of the protocol under real-life data, as well as an analysis of the additional overhead introduced by our protocol.

2 Related work

The idea to randomize the transaction order within a block is folklore in the blockchain space. It has been explored by Yanai [52] and also implemented in the wild [4]. To the best of our knowledge, we are the first to implement the randomization using on-chain randomness and to provide a security analysis for this model. Additionally, Randomspam [4] also acknowledges that some spamming attacks can occur with randomized transactions, where the attacker aims to insert several low-cost transactions to maximize the probability that some of these transactions are positioned exactly at a profitable transaction. Our work reduces the success probability of these attacks by first chunking each transaction into smaller parts and then permuting all chunked transactions, rendering exact positioning attacks less profitable unless more transactions are added, incurring larger gas costs.

A recent line of work [34, 39, 32, 19, 33] formalizes the notion of *fair ordering* of transactions. These protocols ensure, at consensus level, that the final order is consistent with the local order in which transactions are observed by parties. Similarly, the Hashgraph [9] consensus algorithm aims to achieve fairness by having each party locally build a graph with the received transactions. As observed by Kelkar *et al.* [34], a transaction order consistent with the order observed locally for any pair of transactions is not always possible, as Condorcet cycles may be formed. As a result, fair-ordering protocols output a transaction order that is consistent with the view of only some *fraction* of the parties, while some transactions may be output in a batch, i.e., with no order defined among them. Moreover, although order-fairness removes the miner's control over the order of transactions, it does not eliminate front-running and MEV-attacks: a *rushing* adversary that becomes aware of some tx early enough can broadcast its own tx' and make sure that sufficiently many nodes receive tx' before tx.

Another common defense against front-running attacks is the *commit and reveal* technique. The idea is to have a user first commit to a transaction, e.g., by announcing its hash or its encryption, and, once the order is fixed, reveal the actual transaction. However, an adversary can choose not to reveal the transaction, should the final order be non-optimal. Doweck and Eyal [29] employ time-lock puzzle commitments [50], so that a transaction can be brute-force revealed, and protocols such as Unicorn [40] and Bicorn [22] employ verifiable delay functions [15] to mitigate front-running. Whereas they indeed manage to mitigate front running, the main disadvantages of these solutions are threefold: firstly, transactions may be executed much later than submitted, with no concrete upper bound on the revelation time. Secondly, a delay for the time-lock puzzle has to be chosen which matches the network delay and adversary's computational power. Finally, it is unclear who should spend the computational power to solve the time-lock puzzles, especially in proof of work blockchains where this shifts computational power away from mining.

A different line of work [30, 43, 18, 49, 53] hides the transactions until they are ordered with the help of a *committee*. For instance, transactions may be encrypted with the public key of the committee, so that its members can collaboratively decrypt it. However, this method uses threshold encryption [28] and requires a coordinated setup. Also multi-party computation (MPC) has been used [12, 5, 41] to prevent front-running. MPC protocols used in this setting must be tailor-made so that misbehaving is identified and punished [11, 36]. A disadvantage of the aforementioned techniques is that the validity of a transaction can only be checked after it is revealed. These techniques also rely on strong cryptographic assumptions and coordination within the committee. The protocol presented in this work disincentivizes sandwich attacks without requiring hidden transactions or employing computationally heavy cryptography.

Another widely deployed solution against front-running involves a dedicated *trusted third party*. Flashbots[4], Eden[5], and OpenMEV[6] allow Ethereum users to submit transactions to their services, then order received transactions, and forward them to Ethereum miners. Chainlink's Fair Sequencing Service [20], in a similar fashion, aims to collect encrypted transactions from users, totally orders them, and then decrypts them. The third-party service may again be run in a distributed way. The drawback with these solutions is that attacks are not eliminated, but trust is delegated to a different set of parties.

[4] `www.flashbots.net`
[5] `www.edennetwork.io`
[6] `https://openmev.xyz/`

An orthogonal but complementary line of research is taken by Heimbach and Wattenhofer [31]. Instead of eliminating sandwich attacks, they aim to improve the resilience of ordinary transactions against sandwich attacks by strategically setting their slippage tolerance to reduce the risk of both transaction failure as well as sandwich attacks.

Last but not least, Baum *et al.* [10] and Heimbach and Wattenhofer [31] survey the area of front-running attacks.

3 Model

Notation. For a set X, we denote the set of probability distributions on X by $\mu(X)$. For a probability distribution $\nu \in \mu(X)$, we denote sampling x from X according to ν by $x \leftarrow \nu$.

3.1 Block-based atomic broadcast

Parties broadcast transactions and deliver blocks using the events *bab-broadcast*(tx) and *bab-deliver*(b), respectively, where block b contains a sequence of transactions $[tx_1, \ldots, tx_{n_t}]$. The protocol outputs an additional event *bab-mined*(b, P), which signals that block b has been *mined* by party P, where P is defined as the *miner* of b. Notice that *bab-mined*(b, P) signals only the creation of a block and not its delivery. In addition to predicate $VT()$, we also equip our protocol with a predicate $VB()$ to determine the validity of a block. Moreover, we define a function $FB()$, which describes how to fill a block: it gets as input a sequence of transactions and any other data required by the protocol and outputs a block. These predicates and function are determined by the higher-level application or protocol.

▶ **Definition 1.** *A protocol implements* block-based atomic broadcast *with validity predicates $VT()$ and $VB()$ and block-creation function $FB()$ if it satisfies the following properties, except with negligible probability:*

Validity: *If a correct party invokes a* bab-broadcast(tx)*, then every correct party eventually outputs* bab-deliver(b)*, for some block b that contains tx.*

No duplication: *No correct party outputs* bab-deliver(b) *for a block b more than once.*

Integrity: *If a correct party outputs* bab-deliver(b)*, then it has previously output the event* bab-mined$(b, \cdot)$ *exactly once.*

Agreement: *If some correct party outputs* bab-deliver(b)*, then eventually every correct party outputs* bab-deliver(b)*.*

Total order: *Let b and b' be blocks, and P_i and P_j correct parties that output* bab-deliver(b) *and* bab-deliver(b')*. If P_i delivers b before b', then P_j also delivers b before b'.*

External validity: *If a correct party outputs* bab-deliver(b)*, such that $b = [tx_1, \ldots, tx_{n_t}]$, then $VB(b) =$ TRUE and $VT(tx_i) =$ TRUE, for $i \in 1, \ldots, n_t$. Moreover, if $FB(tx_1, \ldots, tx_{n_t})$ returns b, then $VB(b) =$ TRUE.*

Fairness: *There exists $C \in \mathbb{N}$ and $\mu \in \mathbb{R}_{>0}$, such that for all $N \geq C$ consecutive delivered blocks, the fraction of the blocks whose miner is correct is at least μ.*

Observe that the properties assure that *bab-mined*(b, P) is triggered exactly once for each block b, hence each block has a unique miner. For ease of notation, we define on a block b the fields *b.txs*, which contains its transactions, and *b.miner*, which contains its miner. Since blocks are delivered in total order, we can assign them a *height*, a sequence number in their order of delivery, accessible by *b.height*. Finally, for simplicity we assume that a delivered block allows access to all blocks with smaller height, through an array *b.chain*. That is, if $b.height = i$ then $b.chain[i']$ returns b', such that $b'.height = i'$, for all $i' \leq i$.

3.2 Blockchain and network

Blockchain protocols derive their security from different techniques such as *proof of work* (PoW) [45], *proof of stake* (PoS) [27], *proof of space-time* ($PoST$) [24], or *proof of elapsed time* ($PoET$) [16]. In the remainder of the work, we consider a generic protocol Π that has a probabilistic termination condition, capturing all the model above. Furthermore, we model Π as block-based atomic broadcast.

Parties. Similar to previous works, our protocol does not make explicit use of the number of parties or their identities, and does not require the parties themselves to know this number. We assume an static network of n_p parties. We consider the Byzantine model, where f parties may behave arbitrarily, as well as, the rational model where all parties behave maximizing their utilities.

Transactions & Blocks. A transaction tx contains a set of *inputs*, a set of *outputs*, and a number of digital signatures. Transactions are batched into *blocks*. A block contains a number of transactions, n_t, for simplicity we assume n_t to be constant. A block b may contain parameters specific to protocol Π such as references to previous blocks, but we abstract the logic of accessing them in a field $b.chain$, as explained in the context of Definition 1. We allow conditional execution of transactions across blocks, i.e., a transaction can be executed conditioned on the existence of another transaction in a previous block.

Network. A *diffusion functionality* implements communication among the parties, which is structured into *synchronous* rounds. The functionality keeps a $RECEIVE_i$ string for each party P_i and makes it available to P_i at the start of every round. String $RECEIVE_i$ is used to store all messages P_i receives. When a party P_i instructs the diffusion functionality to *broadcast* a message, we say that P_i has *finished its round* and the functionality tags P_i as finished for this round. The adversary, detailed in Section 5, is allowed to read the string of any party at any moment during the execution and to see any messages broadcast by any party immediately. Furthermore, the adversary can write messages directly and selectively into $RECEIVE_i$ for any P_i, so that only P_i receives the message at the beginning of the next round. This models a *rushing* adversary.

When all non-corrupted parties have finished their round, the diffusion functionality takes all messages that were broadcast by non-corrupted parties in the round and adds them to $RECEIVE_i$ for all parties, this is the reason of the name *synchronous* rounds. Every non-corrupted party communicates changes to its local view at the end of each round. If a non-corrupted party creates a block in round r, the new block is received by all parties by round $r+1$. Furthermore, even if the adversary causes a block to be received selectively by only some non-corrupted parties in round r, the block is received by all non-corrupted parties by round $r+2$. The update of the local view also includes the delivery of transactions contained in the blocks that satisfy the conditions to be accepted.

4 Protocol

Our proposed protocol Π^3 ("Partitioned and Permuted Protocol") contains two modifications to a given underlying blockchain protocol Π in order to prevent sandwich MEV attacks.

Our first modification involves randomly permuting the transactions in any given block. Note that a naive way of doing so is to use an external oracle (e.g., DRAND [1] or NIST beacon [35]) to generate the randomness which will be applied to a given block. However,

using an external source of randomness relies on strong trust assumptions on the owners of the source, leaves our protocol vulnerable to a single point of failure and it introduces incentive issues between miners of the chain and owners of the external source. To avoid this, our protocol uses miners of the immediately preceding blocks to generate the randomness. These miners, which we refer to as *leaders* for the given block, are in charge of generating random *partial seeds*. These partial seeds are then combined to form a *seed* which will be the input into a PRG to produce a random permutation that is applied to the transactions in the block. To ensure that the permutation is random, we need first to achieve that leaders participate in the generation of random partial seeds and secondly to ensure the partial seeds generated by the leaders are random. That is, the leaders should not commit to the same partial seed each time or collude with other leaders to generate biased partial seeds. To incentivize each leader to participate in the generation of the seed, Π^3 stipulates that they commit to their partial seed and present a valid opening during the commitment opening period, otherwise their reward will be burned. In typical blockchain protocols, the miner of a block receives the block reward immediately. In Π^3, the miner does not receive the reward until a certain number of additional blocks has been mined. We refer to this as a *waiting phase* and stress that the precise length of the waiting phase is a parameter in our protocol that can be tweaked.

Our second modification is to divide the transfers of each transaction into smaller chunks before permuting the chunked transactions of a block. This modification increases the cardinality of the permutation group in order to reduce the effectiveness of any attack aiming to selectively open partial seeds in order to bias the final permutation.

We stress that our proposed modifications incur minimal computational overhead, since the only possible overhead corresponds to transaction delivery and this aspect is computationally cheap. Thus, the only noticeable impact of our protocol is latency. In the full version [6] we provide an in-depth analysis of the efficiency impact of our proposed modifications.

4.1 Permuting transactions

Protocol Π^3 consists of the following four components (see Figure 1): block mining, generation of the random permutation, reward (re)-distribution, and chunking the transactions.

Appending the partial seeds. Let n_ℓ be the size of the leader set for each block. The miner M_i of block B_i is part of the leader set of blocks B_{i+j}, for $j \in [n_\ell]$. M_i must therefore contribute a partial seed $\sigma_{i,j}$ for each of these n_ℓ blocks following B_i. Hence, M_i needs to create n_ℓ random seeds $\sigma_{i,1}, \ldots, \sigma_{i,n_\ell}$ and commitments to them, $C(\sigma_{i,1}), \ldots, C(\sigma_{i,n_\ell})$. The commitments $C(\sigma_{i,j})$, for $j \in [n_\ell]$, are appended to block B_i, while the seeds $\sigma_{i,j}$ are stored locally by M_i. A block that does not contain n_ℓ commitments is considered invalid.

Looking ahead, we want that any party knowing the committed value can demonstrate it to any other party. Thus, the more standard commitments schemes such as Pedersen commitment [47] are ill-suited. Instead, Π^3 uses a deterministic commitment scheme for committing to permutations, in particular, a collision-resistant cryptographic hash function. When the entropy of the committed values is high enough, then a hash function constitutes a secure commitment scheme. Since the parties commit to a random partial seed, hash functions suffice and yield a cheap commitment scheme.

Opening the commitments. Let $\tau_1, \tau_2 \in \mathbb{N}_{>0}$. Between τ_1 and $\tau_1 + \tau_2$ blocks after the creation of some block B_i, the commitments of the partial permutation to be applied on block B_i must be opened. The miners of these blocks also need to append the openings

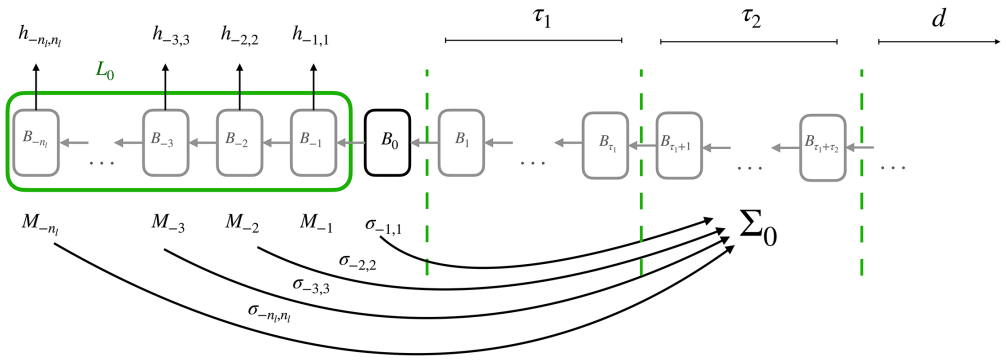


Figure 1 We illustrate the routine for the creation of the random permutation σ_0 created by the leader set L_0 and to be applied on block B_0 in Bitcoin. The leaders L_0 for block B_0 is formed by the miners of the n_ℓ blocks before B, marked with the green box. When party M_i mined block B_i, the party generated some random seed $\sigma_{-i,i}$ and included its commitment $h_{-i,i}$ as part of the newly mined block. After block B_0 is mined, the leaders wait for τ_1 blocks before opening the commitments. The commitments must be included in the following τ_2 blocks. Finally the parties wait until every block containing openings are confirmed before delivering block B_0.

to their blocks, unless a previous block in the chain already contains them (see below for more details). The parameter τ_1 controls the probability of rewriting block B_i after the commitments have been opened. Whereas, parameter τ_2 guarantees that there is enough time for all the honest commitments to be opened and added to some block. Any opening appended a block B_j for $j > i + \tau_1 + \tau_2$ is ignored. We note that specific values of τ_1 and τ_2 might cause our protocol to suffer an increase in latency. We leave these parameters to be specified by the users of our protocol. For the interested reader, we discuss latency-security trade-offs in the full version [6]. The τ_1 blocks created until opening the commitments takes place is known as *silent phase*, whereas the following τ_2 blocks is known as *loud phase*.

A possible way to record the opening of commitments is for the miners that own the commitments to deploy a smart contract that provides a method $open(i, j, \sigma_{i,j})$, where $\sigma_{i,j}$ is a (claimed) opening of the j-th commitment $h_{i,j}$ published in the i-th block B_i. We remark, that the smart contract serves only as proof that an opening to a commitment has been provided, and does not add any functionality to the protocol, so other proof mechanisms can also be considered. The protocol monitors the blockchain for calls to this method. The arguments to each call, as well as the calling party and the block it appears on, are used to determine the final permutations of the blocks and the distribution of the rewards, which we will detail below. We stress that *not* opening a commitment does not impact the progress of protocol, as unopened commitments are ignored.

Deriving the permutation from partial seeds. Let the seed σ_i for block B_i be defined as $\sigma_{i-1,1} \oplus \sigma_{i-2,2} \oplus \ldots \oplus \sigma_{i-n_\ell,n_\ell}$. Given the seed σ_i, let $r_i := G(\sigma_i)$, where $G : \{0,1\}^\lambda \to \{0,1\}^\ell$ is a pseudorandom generator. If at least one of the partial seeds $\sigma_{i,j}$, for $j \in [n_\ell]$, is chosen at random, then σ_i is random as well, and r_i is indistinguishable from a random number [14] without the knowledge of $\sigma_{i,j}$. There are standard algorithms to produce a random permutation from a polynomial number of bits [25].

Incentivizing the behavior. A crucial factor in the security of Π^3 against sandwich MEV attacks is that the permutation used to order transactions within a block should be truly random. Thus, the miners should generate all partial seeds uniformly at random. To

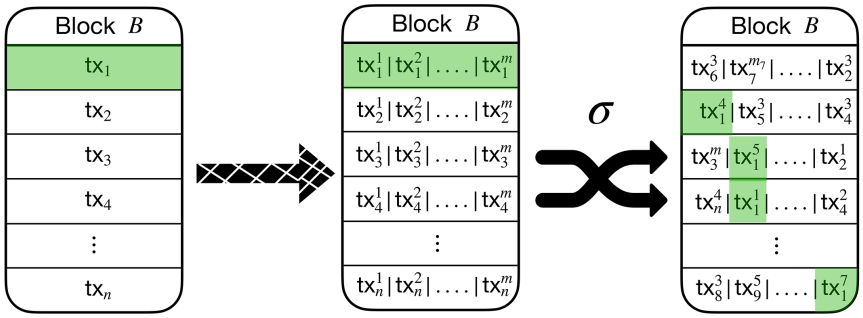


Figure 2 We illustrate the process followed by party M to deliver a block B. We denote by $tx_1, .., tx_n$ the n transaction that constitute B. In the first step, party M breaks each transaction tx_i into m transactions $tx_i^1, .., tx_i^m$ involving a smaller amount. These smaller transactions are later permuted according to the random permutation Σ. Lastly, party M delivers these small transactions in this new order.

incentivize them to do so, we exploit the fact that all leaders remain in the waiting phase for a period of time, which means that they have not yet received the block rewards and fees for mining their block on the blockchain. Note that the waiting phase is $n_\ell + \tau_1 + \tau_2 + d$ blocks long. This implies that their rewards can be claimed by other miners or burned if a party diverges from the proper execution, according to the rules described below. Consider a partial permutation $\sigma_{i,j}$ committed by miner M_i of block B_i. Recall that $\sigma_{i,j}$ will be applied on block B_{i+j} and that miners can be uniquely identified due to the *bab-mined*() event.

1. Before τ_1 blocks have been appended after block B_{i+j} any other leader of the leader set $\mathcal{L}_{i+j}$ who can append a pre-image of $h_{i,j}$ to the chain can receive the reward and fees corresponding to M_i. This mechanism prevents party M_i from disclosing its commitment before every other leader committed its randomness, thus preventing colluding. A miner whose commitment has been discovered by another leader is excluded from all the leader sets.
2. If the opening of $\sigma_{i,j}$ is not appended to any block, miner M_i loses its reward and fees. This mechanism prevents miners from not opening their commitments. Note that miners are incentivized to include all the valid openings, as discussed below.
3. If any of the previous conditions do not apply, party M_i receives an α fraction of the block reward and feed for $\alpha \in (0, 1)$, which would be paid out the moment M_i leaves the waiting phase. Each miner that appends the opening of M_i's commitments gets $\frac{(1-\alpha)\cdot w}{n_\ell}$ for each commitment appended.

In the remainder of this work we refer to the block reward and fees as simply *block reward*.

4.2 Chunking transactions

In all commit-and-open schemes, there exists the vulnerability that malicious parties decide to not open their commitments so as to bias the outcome. In our protocol, any coalition of k leaders can choose between 2^k ways to bias the final permutation. If one miner manages to create multiple blocks out of $B_{i-1}, \ldots, B_{i-n_\ell}$, this does not even require collusion with others.

The adaptive attacks mounted through withholding can be countered with a simultaneous broadcast abstraction [23], but realizing this is almost impossible in practice [37], especially in the blockchain domain. Alternatively, time-lock puzzles may negate the effect of the

delay. But this technique costs computational effort, which may have a negative impact on the environment and possibly also on the protocol. In the particular case of generating a permutation, there is an alternative.

Chunking transfers in transactions. Let us assume that a block contains n_{tx} transactions, this means there exist $n_{tx}!$ possible permutations of them. A coalition of k leaders can choose between 2^k possible permutations among the $n_{tx}!$ total permutations. Furthermore, in the simplest case the coalition only aims to order the three transactions that constitute the sandwich attack, thus the fraction of advantageous permutations is $\frac{1}{6}$, the fraction of disadvantageous permutations is $\frac{1}{6}$ and the remaining ones are neutral. If k is big enough, the coalition could still extract enough value to compensate for the lost block rewards of those parties that do not open their commitment.

Therefore we want to increase the size of the permuted space. We assume here that every transaction consists of arbitrary code and a few specialized instructions that are *transfers* of coins or tokens. These may be the *native payment operation* of the blockchain or operations that involve a well-known *standard format for tokens*, which are emulated by smart contracts (such as the ERC-20 standard in Ethereum). We now divide every transfer generated by some transaction into m chunks.

For instance, suppose transaction tx_i consists of Alice paying Bob 1 ETH. In our protocol, each party would locally split tx_i into m chunks $tx_i^1, \dots tx_i^m$, consisting of Alice transferring $1/m$ ETH to Bob each. After all transactions are chunked, the permutation will be applied to the larger set of transactions. There exist $(n_{tx}m)!$ permutations and the coalition would need to order the $3m$ chunks that constitute the involved transactions. Furthermore, for a given permutation with some chunks ordered beneficially, there will exist chunks ordered in a disadvantageous way, with overwhelming probability. The coalition needs to optimize the good ordering of some chunks while keeping the bad ordering under control. Obtaining a favorable ordering becomes extremely unlikely as number of chunks m grows (Section 5.2).

Execution of transactions and chunks. Transactions contain arbitrary code whose execution produces an ordered list of transfers, as introduced before. The process of chunking proceeds in two stages.

In the first stage, the party executes the code of all transactions contained in the block serially, in the order determined by the miner; this produces a list of transfers and the corresponding amounts for each transaction. Some transactions may turn out to be invalid, they are removed from the further processing of the block.

In the second stage, for each valid transaction tx a list of m transfer chunks $tx^1, \dots tx^m$ is produced such that tx^1 contains the *code* executed by tx, and its transfers have their amount set to $1/m$ of the number computed by the code. The transaction chunks $tx^2, \dots tx^m$ contain *only* the transfers, with their numbers set to $1/m$ of original amounts, but these transactions do not execute further code. If the code executed by tx^1 produces different transfers than in the first stage, the execution of tx^1 is aborted, also the execution of $tx^2, \dots tx^m$. We consider two transfers to be the same if they transfer the same amount of coins or tokens from the same source address to the same destination address. Note the blockchain state produced by a transaction in the first stage can differ from the state produced by the same transaction in the actual execution according to the second stage execute; this is the case, for instance, when it interacts with a smart contract.

The block being permuted now contains up to m times as many chunks as the original block contained transactions, each of them transferring $1/m$ the value.

Notice that the permutation is not uniformly random across all choices, but needs to respect that tx^1 appears first within the set of chunks resulting from tx. However, this restriction in the permutation does not constitute any loss of generality since every chunk performs an identical transfer. An adversarial miner can utilize fine-grained conditions such as slippage to additionally control the conditional execution of transactions – and in our case transaction chunks – in a given block. The execution of transactions explained above guarantees atomicity: all chunks are executed or no chunk is executed. In the full version [6] we present an in-depth analysis of how slippage could lead to higher expected revenue, which may also be of independent interest.

4.3 Details

In Algorithm 1 we show the pseudocode for protocol Π^3, which implements a block-based atomic broadcast (*bab*) primitive. The pseudocode assumes an underlying protocol Π, which is also modeled as a block-based atomic broadcast (*bab*) primitive, as defined in Section 3. The user or high-level application interacts with Π^3 by invoking Π^3-*broadcast*(tx) events. These are handled by invoking the corresponding Π-*broadcast*(tx) event on the underlying protocol Π (L3-4).

Protocol Π outputs an event *bab-mined*(b, Q) whenever some party Q mines a new block b (L5). For Π^3, the mining of a new block at height i starts the opening phase for the block at height $i_{\text{open}} = i - \tau_1 - 1$ (L6). Hence, party P loops through the n_ℓ blocks before i_{open} and checks whether it is the miner of each of them (L7-9). If this is the case, P must provide a valid opening to the commitment related to block at height i_{open}. The opening is achieved by a specific type of transaction, for example through a call to a smart contract. In the pseudocode we abstract this into a function *Open*().

Protocol Π outputs an event Π-*deliver*(b) whenever a block b is delivered (L10). According to the analysis of our protocol, this will allow Π^3 to deliver the block $\tau_1 + \tau_2$ positions higher than b, i.e., the block b_{del} at height $i_{\text{del}} = b.height - \tau_1 - \tau_2$. To this goal, Π^3 first reads the commitments related to b_{del} (L12-13). By construction of Π^3, a commitment $c_{i,j}$, written on block b_i, is used to order the transactions in block b_{i+j}. Hence, the commitments related to b_{del} have been written on the n_ℓ blocks before b_{del}. Protocol Π^3 then reads the openings to these commitments (L14-17). Again by construction of Π^3, the openings of the commitments related to b_{del} have been written on the blocks with height $i_{\text{del}} + \tau_1 + 1$ to $i_{\text{del}} + \tau_1 + \tau_2$. For each of these blocks, Π^3 loops through its transactions that contain an opening. L16 then checks whether the opening is for a commitment related to block b_{del} and whether the opening is valid. Protocol Π^3 then calculates the final permutation Σ to be applied to block b (L18-22). As presented in Section 4.1, $\Sigma = \mathit{PermFromRandBits}(G(\mathit{seed}))$, where *seed* is the XOR of all valid openings for block b, G is a pseudorandom generator, and *PermFromRandBits* an algorithm that derives a permutation from random bits. The remaining of this block chunks the transactions contained in b (L23-25), applies Σ on the chunked transactions (L26), and swaps the first permuted chunk of each of each transaction with the chunk containing the code (L28). The function *Chunk*() is explained in Section 4.2. Finally, Π^3 delivers block b containing the chunked and permuted transactions through the Π^3-*deliver*(b) event (L30).

The function *FB*() is an upcall from block-based atomic broadcast. It specifies how a block is filled with transactions and additional data. For simplicity, the pseudocode omits any detail specific to *bab*. It first writes all given transactions on the block, then picks uniformly at random n_ℓ bit-strings of length λ. These are the partial random seeds to be used in the permutation of the following n_ℓ blocks, if the block that is currently being built gets mined and delivered by *bab*. The commitments to these partial seeds are appended on the block.

Algorithm 1 Protocol Π^3. Code for party P.

Implements: Protocol Π^3
Uses: block-based atomic broadcast Π

State:
$\sigma[i,j] \leftarrow \perp$, for all $i \geq 1, j \in [n_\ell]$
$c[i,j] \leftarrow \perp$, for all $i \geq 1, j \in [n_\ell]$

upon event $\langle \Pi^3\text{-}broadcast, tx \rangle$ **do**
  invoke $\langle \Pi\text{-}broadcast, tx \rangle$

upon event $\langle \Pi\text{-}mined, b, Q \rangle$ **do**
  $i_{\text{open}} \leftarrow b.height - \tau_1 - 1$
  for $i' \in [i_{\text{open}} - n_\ell - 1, i_{\text{open}} - 1]$ **do**
    if $b.chain[i'].miner = P$ **then**
      $Open(b.chain[i'].commitments[i_{\text{open}} - i'])$

upon event $\langle \Pi\text{-}deliver, b \rangle$ **do**
  $i_{\text{del}} \leftarrow b.height - \tau_1 - \tau_2$
  for $j \in [n_\ell]$ **do** // Read commitments for block b
    $c[i_{\text{del}}, j] \leftarrow b.chain[i_{\text{del}} - j].commitments[j]$
  for $i' \in [i_{\text{del}} + \tau_1 + 1, i_{\text{del}} + \tau_1 + \tau_2]$ **do** // Read the openings for block b
    for $tx \in b.chain[i'].txs$ **such that** $tx = open(k, l, \sigma)$ **do**
      if $k + l = i_{\text{del}}$ **and** $H(\sigma) = c[i_{\text{del}}, l]$ **then**
        $\sigma[i_{\text{del}}, l] \leftarrow \sigma$
  $seed \leftarrow 0^\lambda$
  for $j \in [n_\ell]$ **do** // Compute final permutation for block b
    if $\sigma[i_{\text{del}}, j] \neq \perp$ **then**
      $seed \leftarrow seed \oplus \sigma[i_{\text{del}}, j]$
  $\Sigma \leftarrow PermFromRandBits(G(seed))$
  $chunked\_txs \leftarrow [\,]$
  for $tx \in b.txs$ **do** // Chunk and permute transactions in block b
    $chunked\_txs \leftarrow chunked\_txs \| Chunk(tx, m)$
  $chunks \leftarrow Permute(\Sigma, chunked\_txs)$
  $chunks \leftarrow SwapChunks(chunks)$ // For each tx, swap the first chunk in $chunks$ with tx^1
  $b.txs \leftarrow chunks$
  invoke $\langle \Pi^3\text{-}deliver, b \rangle$

function $FB(txs)$ **:**
  $data \leftarrow [\,]$
  for $tx \in txs$ **do**
    $data \leftarrow data \| tx$
  for $j \in [n_\ell]$ **do**
    $\sigma \xleftarrow{\$} \{0,1\}^\lambda$
    $c \leftarrow H(\sigma)$
    $data \leftarrow data \| c$
  return $data$

function $VB(b)$ **:**
  if $\big(\exists tx \in b.txs : \neg VT(tx)\big) \vee \big(\exists j \in [n_\ell] : b.commitments[j] = \perp\big)$ **then**
    return FALSE
  return TRUE

Finally, the predicate $VB()$ specifies that a block is valid if all its transactions are valid, as specified by $VT()$, and if it contains n_ℓ commitments. The predicate $VT()$ is omitted, as its implementation does not affect Π^3.

5 Analysis

5.1 Security analysis

We model the adversary as an interactive Turing machine (ITM) that corrupts up to t parties at the beginning of the execution. Corrupted parties follow the instructions of the adversary and may diverge arbitrarily from the execution of the protocol. The adversary also has control over the *diffusion functionality*. That is, she can schedule the delivery of messages (within the Δ rounds), as well as read the $RECEIVE_i$ of every party at any moment of the execution and directly write in the $RECEIVE_i$ of any party.

We first show that the security of our construction is derived from the security of the original protocol. Given an execution of protocol Π^3, we define the equivalent execution in protocol Π as the execution in which every party follows the same steps but the commitment, opening, and randomization of transactions are omitted. We also recall the parameters τ_1 and τ_2 that denote the length (in blocks) of the silent and loud phase respectively.

▶ **Lemma 2.** *The probability that an adversary can rewrite a block after any honest partial permutations have been opened is negligible in* τ_1.

Proof. Assume an adversary controlling up to t parties and a block B. We know that if $\tau_1 > d$, protocol Π would deliver block B, thus an adversary cannot revert the chain to modify the order of the transactions stored in B but with negligible probability. ◀

▶ **Lemma 3.** *The probability that an adversary can rewrite a chain omitting the opening of some honest partial permutation is negligible in* τ_2.

Proof. The fairness quality of protocol Π states that for any consecutive N blocks, if $N \geq N_0$ the fraction of honest blocks is at least μ. Thus, if $\tau_2 \geq \max\{N_0, \frac{1}{\mu}\}$, there exists at least one honest block containing every opening that is not previously included in the chain. Since ◀

Our construction aims to turn any protocol into a protocol robust against sandwich attacks. However, there might be new vulnerabilities. Intuitively, our construction should not introduce any vulnerability because the only modified aspect is the order in which transactions are delivered. Theorem 5 formalizes this intuition.

▶ Remark 4. Note that every Π-*delivered* block is also Π^3-*delivered* some block after (Line10–30). Note also that every Π^3-*delivered* is also Π*delivered*. Furthermore, the blocks are delivered in the same order.

▶ **Theorem 5.** *If protocol* Π *implements block-based atomic broadcast, then the Partitioned and Permuted Protocol* Π^3 *implements block-based atomic broadcast.*

Proof. According to Remark 4, the set of Π^3-*delivered* blocks is the same as the set of Π-*delivered* blocks.

Validity. Assume that an honest party Π^3-*broadcasts*(tx) transaction tx. The party first Π-*broadcasts*(tx) (L3–4). The validity property of protocol Π guarantees that eventually a block b containing transaction tx is Π-*delivered*. According to Remark 4, the honest party eventually Π^3-*delivers* a block containing tx and Π^3 satisfies the validity property of block-based atomic broadcast.

No-duplication. Note that Π^3 delivers the same set of blocks as protocol Π, Remark 4. Thus, the no-duplication property of protocol Π^3 is inherited directly from the no-duplication of protocol Π.

Agreement. Consider two honest parties P and Q such that party P Π^3-*delivers* block b. Remark 4 guarantees that P alsoΠ-*delivers* block b. The agreement property of protocol Π ensure that Q eventually Π-*delivers* block b. Remark 4 guarantees that Q eventually Π^3-*delivers* block b. Note that the block b delivered by both P and Q may differ in how the transactions are chunked and permuted. However, Lemmas 2 and 3 guarantee all correct parties agree on the same permutation with all but negligible probability. Hence, we conclude that protocol Π^3 satisfies the agreement property.

Total order. Remark 4 guarantees that the order in which any honest party Π^3-*delivers* two block b_1 and b_2 is the same as it Π-*delivers* them. Thus, the total order property of protocol Π guarantees the total order property of protocol Π^3.

External validity. This follows from the external validity of Π.

Fairness. According to Remark 4 the same blocks and in the same order are both Π^3-*delivered* and Π-*delivered*. Hence, the fairness property of Π^3 is inherited from the fairness property of protocol Π. ◀

After showing that Π^3 is as secure as the original protocol Π. We turn our attention to analyzing the behavior of Π^3 under sandwich attacks, in the upcoming section.

5.2 Game-theoretic analysis

Here, we aim to show that if we assume all miners are rational, i.e., they prioritize maximizing their own payoff, behaving honestly as according to our protocol Π^3 is a stable strategy.

Strategic games. For $N \in \mathbb{N}$, let $\Gamma = (N, (S_i), (u_i))$ be an N party game where S_i is a finite set of strategies for each party $i \in [N]$. Let $S := S_1 \times \cdots \times S_N$ denote the set of outcomes of the game. The utility function of each party i, $u_i : S \to \mathbb{R}$, gives the payoff of party i given an outcome of Γ. For any party i, a *mixed strategy* s_i is a distribution in $\mu(S_i)$. A *strategy profile* of Γ is $s := s_1 \times \cdots \times s_N$ where s_i is a mixed strategy of party i. The *expected utility* of a party i given a mixed strategy profile s is defined as $u_i(s) = \mathbb{E}_{a_1 \leftarrow s_1, \cdots, a_N \leftarrow s_N}[u_i(a_1), \cdots, u_i(a_N)]$. Finally, we note that if s_i is a Dirac distribution over a single strategy $a_i \in S_i$, we say s_i is a *pure strategy* for party i.

Notation. Let w denote the total reward for mining a block and q the negligible probability that a PPT adversary guesses a correct opening. Recall in Section 4.1 that the total block reward w is split between the miner of the block who gets $\alpha \cdot w$ and the miners that append the correct openings who get $\frac{(1-\alpha)\cdot w}{n_\ell}$ for each correct opening they append. For a given block, we denote by m the number of chunks for *each* transaction in the block, and by λ the utility of the sandwich attack on the block. Specifically, λ refers to the utility of a sandwich attack performed on the original transactions in the order they are in *before* chunking and permuting them. We also denote the optimal sandwich utility by Λ, which is the maximum utility one can get by performing a sandwich attack. Finally, we denote by $\hat{\lambda}_i$ the *average* utility of the sandwich attack taken over all blocks on the chain for a specific miner M_i. This can be computed easily as the transaction mempool is public. We stress that it is important to look at the average sandwich utility for each miner separately and not the average over all miners as the utility a miner can derive from a sandwich attack depends on their available liquidity (i.e., how much assets they can spare to front-run and back-run the transactions).

Quasi-strong ε-Nash Equilibrium. In terms of game theoretic security, we want our protocols to be resilient to deviations of any subset of miners that form a coalition and deviate jointly. The security notion we want to achieve is that of an *quasi-strong ε-Nash Equilibrium* [8, 13, 21]. Let C denote the coalition of players. For any strategy profile s, we denote by $u_C(s)$ the expected utility of the coalition under s. We denote by $u_C(s'_C, s_{-C})$ the expected utility of the coalition when playing according to some other strategy profile s'_C given the other players that are not part of the coalition play according to s.

▶ **Definition 6** (Quasi-strong ε-Nash Equilibrium). *A quasi-strong ε-Nash Equilibrium is a mixed strategy profile s such that for any other strategy profile s'_C, $u_C(s) \geq u_C(s'_C, s_{-C}) - \varepsilon$ for some $\varepsilon > 0$.*

The notion of a quasi-strong Nash Equilibrium is particularly useful in the context of blockchains as the coalition could potentially be controlled by a single miner with sufficient resources [21]. The notion of an ε-equilibrium is also important in cases where there could be a small incentive (captured by the ε parameter) to deviate from the protocol, and of course the smaller one can make ε, the more meaningful the equilibrium.

Subgame perfection. We also consider games that span several rounds and we model them as extensive-form games (see, e.g., [46] for a formal definition). Extensive form games can be represented as a game tree tx where the non-leaf vertices of the tree are partitioned to sets corresponding to the players. The vertices belonging to each player are further partitioned into information sets I which capture the idea that a player making a move at vertex $x \in I$ is uncertain whether they are making the move from x or some other vertex $x' \in I$. A subgame of an extensive-form game corresponds to a subtree in tx rooted at any non-leaf vertex x that belongs to its own information set, i.e., there are no other vertices that are the set except for x. A strategy profile is a *quasi-strong subgame perfect ε-equilibrium* if it is a quasi-strong ε-Nash equilibrium for all subgames in the extensive-form game.

The induced game. Let us divide our protocol into epochs: each epoch is designed around a given block say B_i and begins with the generation of random seeds for B_i and ends with appending the openings for the committed random seeds for B_i (i.e., block $B_{i+\tau_1+\tau_2}$). We define the underlying game Γ induced by any given epoch of our protocol Π^3. Γ is a $(\tau_2 + 1)$-round extensive form game played by $n_\ell + \tau_2$ parties (n_ℓ leaders comprising the leader set L_i for any block B_i and the τ_2 miners that mine the blocks $B_{i+\tau_1+1} \dots B_{i+\tau_1+\tau_2}$). Note that although we have $\binom{N}{\tau_2}$ sets of τ_2 miners to choose from (where N is the total number of miners in the chain) to be the miners of the blocks $B_{i+\tau_1+1} \dots B_{i+\tau_1+\tau_2}$, we can simply fix any set of τ_2 miners together with L_i to be the parties of Γ as we assume all miners are rational and so the analysis of the utilities of any set of τ_2 miners will be the same in expectation. We use A to denote the set of all miners in τ_2. In what follows, we assume an arbitrary but fixed ordering of the miners in A. Round 1 of Γ consists of only the parties in L_i performing actions, namely picking a random seed and committing to it. In rounds $2, \dots, \tau_2 + 1$ of Γ, each member of L_i can act by choosing to open their commitment or not. However, the moment a member of L_i opens its commitment in a given round, they lose the chance to open their commitment in any subsequent round. Only one miner from A and according to the imposed ordering acts in each round from round 2 to $\tau_2 + 1$ of Γ. The choice of actions of the miner in any of these rounds are the subsets of the set of existing commitment openings (from members of L_i) to append to their block. Finally we note that the $L_i \cap A$ is not necessarily empty and thus miners in the intersection can choose to open and append their commitment in the same round.

Let us define the honest strategy profile as the profile in which all members of L_i choose to generate a random seed in round 1 of Γ, all members of L_i open their commitments at round τ_2 (i.e., at block $B_{i+\tau_1+\tau_2-1}$), and each member of A appends all existing opened commitments that appear in the previous round. We denote the honest strategy profile by s. The security notion we want to achieve for our protocol is a quasi-strong subgame perfect ε-equilibrium (refer to Definition 6). Looking ahead, we will also prove that ε can be made arbitrarily small by increasing the number m of chunks.

▶ **Lemma 7.** *The expected utility of an honest leader is at least* $(1-q)^{n_\ell}\alpha w$.

Proof. The expected utility for a user following the honest strategy comes from the sum of the block reward, the expected utility from the ordering of any of their transactions within the block, and appending valid openings of committed seeds (if any) to their blocks. The expected utility from the ordering of transactions is 0 due to symmetry: each possible order is equally likely, for each order that gives some positive utility, there exists a different order producing the same negative utility. The expected utility from the block reward is $(1-q)^{n_\ell}\alpha w$. Thus, the total expected utility of an honest miner is at least $(1-q)^{n_\ell}\alpha w$. ◀

We outline and analyze two broad classes of deviations or attacks any coalition can attempt in this setting. The first class happens at round 1 of Γ where the members of the coalition commit to previously agreed seeds to produce a specific permutation of the transactions. The coalition then behaves honestly from round 2 to τ_2+1 of Γ. We call this attack the *chosen permutation attack* and denote this attack strategy by s_{CP}. In the second class, the coalition behaves honestly at round 1 of Γ, but deviates from round 2 onwards where some members selectively withhold opening or appending commitments to bias the final permutation. We call this attack the *biased permutation attack*, and denote it by s_{BP}.

Chosen permutation attack. Before we describe and analyze the chosen permutation attack (for say a block B_i), we first show that a necessary condition for the attack to be successful, that is, the coalition's desired permutation happens almost surely, is that at least all n_ℓ leaders in L_i have to be involved in the coalition (members of A can also be involved in the coalition, however as we will show this will simply increase the cost). To do so, we let S denote the set of permutations over the list of transactions and their chunks, and we define what we mean by a protocol Π_{perm} (involving n parties) outputs random a permutation in S by the following indistinguishability game called *random permutation indistinguishability* played between a PPT adversary, a challenger, and a protocol Π_{perm}. First, the adversary corrupts up to $n-1$ parties. The adversary has access to the corrupted parties' transcripts. Then, the challenger samples σ_0 uniformly at random from S, and sets σ_1 to be the output of Π_{perm}. After that, the challenger flips a random bit b and sends σ_b to the adversary. The game ends with the adversary outputting a bit b'. If $b'=b$, the adversary wins the game. We say a protocol Π_{perm} outputs a random permutation if the the adversary wins the above game with probability $\frac{1}{2}+\varepsilon$ for some negligible ε. Let us define the output of a single round of Π^3 as the random permutation that is generated from the seeds generated from all leaders in the round according to the algorithm described in Section 4.1. The following lemma states that as long as a single leader is honest, the output of Π^3 is pseudorandom.

▶ **Lemma 8.** *An adversary that corrupts at most* $n_\ell - 1$ *leaders in a single round of* Π^3 *can only win the random permutation indistinguishability game with negligible probability.*

Proof. The proof follows in the same way as introduced by M. Blum [14], with the addition of the PRG. ◀

Lemma 8 implies that launching the chosen permutation attack and thus choosing to deviate at round 1 of Γ comes with an implicit cost: either a single miner has to mine n_ℓ blocks in a row so the miner single-handedly forms the coalition, or *all* leaders in L_i have to be coordinated into playing according to a predefined strategy.

▶ **Lemma 9.** *Given the underlying blockchain is secure, the expected utility of the single miner when playing according to s_{CP} is at most $\frac{\lambda}{2^{n_\ell}}$ more than the expected utility of following the honest strategy.*

Proof. Since the underlying blockchain is secure, a necessary condition is that a single miner cannot own more than $\frac{1}{2}$ of the total amount of resources owned by all miners of the protocol. Thus, the probability of mining n_ℓ blocks in a row is strictly less than $\frac{1}{2^{n_\ell}}$. This means that the expected utility under the attack strategy $u_C(s_{CP}) < \frac{\lambda}{2^{n_\ell}} + n_\ell \alpha w$, which is at most $\frac{\lambda}{2^{n_\ell}}$ larger than the expected utility under the honest strategy which is $u_C(s) = n_\ell \alpha w$. ◀

The attack strategy of a coalition composed by more than one miner is more complex compared to the case where there is a single miner, as the coalition needs to ensure its members coordinate strategies. First, the coalition works with the miner of block B_i to select and fix a permutation generated by a specific PRG seed σ_i. Then, the coalition secret shares σ_i among its members[7]. After that, the coalition sets up some punishment scheme to penalize members that do not reveal their partial seeds[8]. Finally, the coalition commits and reveals these partial seeds in accordance to the protocol Π^3. Let $\mathcal{C}$ denote the expected cost of coordinating the whole chosen permutation attack for the coalition. For this attack to succeed, the expected coordination cost has to be smaller than the expected profit λ.

▶ **Lemma 10.** *The chosen permutation attack fails to be profitable compared to the honest strategy if $\mathcal{C} > \lambda$.*

Proof. From Lemma 7, the expected revenue of an honest miner is $(1-q)^{n_\ell}\alpha w$, thus the expected revenue of the coalition when following the honest strategy is $u_C(s) = n_\ell \cdot (1-q)^{n_\ell}\alpha w$. The expected revenue for the chosen permutation attack strategy is $u_C(s_{CP}) = n_\ell \cdot (1-q)^{n_\ell}\alpha w + \lambda - \mathcal{C}$. Thus, assuming $\mathcal{C} > \lambda$, and since the expected revenue from a mixed strategy is a convex combination of the revenues of the honest and attack strategies, the pure honest strategy gives a strictly larger expected payoff compared to any mixed strategy. ◀

▶ Remark 11. Computing, or even estimating, the coordination cost is non-trivial as it consists of several dimensions and also depends on a myriad of factors and assumptions. A few notable costs are, firstly, timing costs. The coalition has to convince and coordinate all the leaders to agree on a permutation and also commit and reveal them during a short interval of d blocks. This involves the cost of securely communicating with all the leaders and also the computational cost involved in setting up the secret sharing scheme. A second factor is the choice of the initial order of transactions, which the coalition would have to also agree on with the miner of the attacked block. Picking transactions greedily would be the simplest choice as finding the optimal set of transactions from the mempool is NP-hard [42]. Finally, the coalition has to set up a punishment scheme to penalize members that do not

[7] This not only prevents members from knowing the partial seeds of other members and hence stealing their block reward, but also additionally safeguards the partial seeds of the members against the miner of block B_i who cannot generate a partial seed of their block and hence has nothing to lose.

[8] This ensures that every member will reveal reveal their partial seeds and the permutation will be generated properly.

reveal their permutations. If we ignore the cost of setting up such a scheme, this can be implemented using a deposit scheme with the size of the deposit at least the value of the expected additional per user profit from the sandwich attack [51]. This implies an opportunity cost at least linear in $\frac{\lambda}{n_\ell}$, as well as the assumption that each member has at least $\frac{\lambda}{n_\ell}$ to spare to participate in the attack. Additionally, we note that the coalition could extend to miners from A which are outside the leader set L_i. However, since these miners do not contribute to generating the random seeds, they simply add to the communication cost of the coalition. Finally, we note that the coordination cannot be planned in advance due to the unpredictability of the block mining procedure.

Biased permutation attack. The intuition behind this attack is that any coalition that controls $k \leq n_\ell$ commitments can choose to select the ones to open or append, which allows the coalition to chose among 2^k possible permutations in order to bias the final ordering. This can be achieved in two situations: either k out of n_ℓ leaders of L_i form a coalition and decide which of their commitments to open, or some subset of miners in the loud phase (of size say $\tilde{k}$) form a coalition and end up controlling k openings, let $\kappa := \min\{k, \tilde{k}\}$. Unlike in the case of the chosen permutation attack, it suffices consider the case where we have a *single* miner that happens to either occupy k leader positions among the group of leaders L_i or mine the $\tilde{k}$ blocks that belong to the coalition in the loud phase. This is because the case where a coalition of distinct miners that collude only adds additional coordination cost. The probability that any such coalition gains any additional utility by performing the biased permutation attack compared to the honest strategy can be upper-bounded. Let *revenue* denote the utility the coalition would gain from performing the biased permutation attack.

▶ **Lemma 12.** *The probability that a coalition of κ members performing the biased permutation attack achieves utility of at least $\kappa w > 0$ is* $\mathrm{P}[\textit{revenue} \geq \kappa w] \leq 1 - (1 - e^{-\frac{2m\kappa w}{\lambda}})^{2^k}$.

Proof. Given a random permutation and a sandwich attack with original utility λ (utility if the order of the transactions were not randomized), denote by $\{X_i(\sigma)\}_{i=1}^m$ the utility produced by chunk i. The sum of these random variables $X(\sigma) = \sum_{i=1}^n X_i(\sigma)$ represents the total utility of a sandwich attack (after chunking and permuting). X takes values in $[-\lambda, \lambda]$, thus the variables $\{X_i(\sigma)\}$ take values in $[-\frac{\lambda}{m}, \frac{\lambda}{m}]$, are equally distributed and are independent. We define the random variables $Y_i(\sigma) = X_i(\sigma) + \frac{\lambda}{m} \in [0, \frac{2\lambda}{m}]$, and $Y(\sigma) = \sum_{i=1}^n Y_i(\sigma) \in [0, 2\lambda]$. Using lemma 7, $\mathrm{E}[Y_i(\sigma)] = \frac{\lambda}{m}$ and $\mathrm{E}[Y(\sigma)] = \lambda$. Applying Chernoff's bound [44] to Y,

$$\mathrm{P}[Y(\sigma) \geq (1+\delta)\mathrm{E}[Y(\sigma)]] \leq e^{\frac{-2\delta^2 \mathrm{E}[Y(\sigma)]^2}{m(\frac{\lambda}{m})^2}} = e^{-2m\delta^2} \tag{1}$$

for $\delta > 0$. We can rewrite Equation 1 as follows:

$$\mathrm{P}[\textit{revenue}(\sigma) \geq \delta\lambda] = \mathrm{P}[\textit{revenue}(\sigma) + \lambda \geq (1+\delta)\lambda] = \mathrm{P}[Y(\sigma) \geq (1+\delta)\mathrm{E}[Y(\sigma)]] \leq e^{-2m\delta^2}.$$

Using the law of total probability we obtain that $\mathrm{P}[\textit{revenue}(\sigma) \leq \delta\lambda] \geq 1 - e^{-2m\delta^2}$. Considering the maximum over the 2^k possible permutations σ and $\delta = \frac{\kappa\omega}{\lambda}$ we conclude that

$$\mathrm{P}[\textit{revenue} \geq \kappa w] = 1 - \mathrm{P}[\textit{revenue} \leq \kappa w] = 1 - \mathrm{P}[\textit{revenue}(\sigma) \leq \kappa w] \leq 1 - (1 - e^{-\frac{2m\kappa w}{\lambda}})^{2^k}. \blacktriangleleft$$

▶ **Lemma 13.** *The probability that a coalition of κ members has positive additional utility is:* $\mathrm{P}[\textit{revenue} \geq 0] \leq \max_{k' \leq \kappa} \left\{1 - (1 - e^{-\frac{2mk'w}{\lambda}})^{2^k}\right\}$.

Proof. Lemma 12 states a bound for the probability that a coalition of κ parties has a utility of at least $\kappa w > 0$, the penalty for not opening κ commitments. Thus, the general case for a coalition aiming to maximize profit is the maximum over $k' \leq \kappa$. ◀

Recall that Λ is the maximal utility and let $p_{k,\lambda}$ denote $\max_{k' \leq \kappa}\{1-(1-e^{-\frac{2m(1-q)^{n_\ell}k'w}{\lambda}})^{2^k}\}$. Then, the expected additional utility from the biased permutation attack of a single miner controlling k leaders is no greater than $p_{k,\lambda}\Lambda$.

Lemmas 10,12 and 13 allow us to prove our main theorem.

▶ **Theorem 14.** *Suppose $\mathcal{C} > \lambda$, then the honest strategy $s = ((\text{random seed})_{i=1}^{n_\ell}, (\text{open})_{i=1}^{n_\ell})$ is a quasi-strong subgame perfect ε-equilibrium in Γ for $\varepsilon = \max\{\frac{\lambda}{2^{n_\ell}}, p_{k,\lambda}\Lambda\}$.*

Proof. We first observe that the expected utility of a coalition that mixes both the chosen and biased permutation attack strategies is no greater than the expected utility of a coalition that performs the chosen permutation attack with a different chosen permutation that accounts for the biasing of the permutation in the second round of Γ. Hence, it suffices to analyze the expected utility of the coalition when implementing either of these strategies, i.e., deviating at round 1 of Γ or from rounds 2 onwards.

We first analyze the expected utility of a coalition when implementing the chosen permutation attack, which occurs at round 1 or Γ. Since we assume $\mathcal{C} > \lambda$, from Lemma 9 and Lemma 10, we see that any additional expected payoff of any coalition that deviates only at round 1 of Γ by implementing the chosen permutation attack compared to the expected revenue of behaving honestly is at most $\frac{\lambda}{2^{n_\ell}}$.

Now we analyze the expected utility of a coalition when implementing the biased permutation attack. From Lemmas 12 and 13, we see that the strategy that implements the biased permutation attack across all of rounds 2 to $\tau_2 + 1$ of Γ only gives at most $p_{k,\lambda}\Lambda$ more payoff in expectation compared to following the honest strategy s in these rounds.

As such, if we set $\varepsilon = \max\{\frac{\lambda}{2^{n_\ell}}, p_{k,\lambda}\Lambda\}$ to be the largest difference in additional expected revenues between both strategies, we see that $s = ((\text{random seed})_{i=1}^{n_\ell}, (\text{open})_{i=1}^{n_\ell})$ is a quasi-strong ε-subgame perfect equilibrium of Γ. ◀

▶ Remark 15. Recall that ε bounds the additional expected utility an adversary can gain by deviating from the honest strategy profile s. The security of our protocol therefore improves as $\varepsilon = \max\{\frac{\lambda}{2^{n_\ell}}, p_{k,\lambda}\Lambda\}$ decreases. We observe that the first component $\frac{\lambda}{2^{n_\ell}}$ goes to 0 exponentially as the size of the leader set n_ℓ increases. As for the second component $p_{k,\lambda}$, we conduct an empirical analysis of sandwich attacks on Ethereum, see full version [6], to estimate $p_{k,\lambda}$ and we show that this value approaches zero as the number of chunks m increases.

6 Conclusion

In this paper we introduced a new construction that can be implemented on top of any blockchain protocol with three main properties. First, the construction does not add any vulnerability to the old protocol, i.e., the security properties remain unchanged. Secondly, performing sandwich attacks in the new protocol is no longer profitable. Thirdly, the construction incurs in minimal overhead with the exception of a minor increase in the latency of the protocol. Our empirical study of sandwich attacks on the Ethereum blockchain also validates the design principles behind our protocol, demonstrating that our protocol can be easily implemented to mitigate sandwich MEV attacks on the Ethereum blockchain.

References

1 Distributed randomness beacon, 2023. URL: `https://drand.love/`.
2 Ethereum, 2023. URL: `https://ethereum.org/en/whitepaper/`.

3 MEV over time, 2023. URL: https://explore.flashbots.net/.

4 Random ordering of equally-priced transactions incentivises competitive spam, 2023. URL: https://github.com/ethereum/go-ethereum/issues/21350.

5 Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder - scalable, robust anonymous committed broadcast. In *CCS*, pages 1233–1252. ACM, 2020. doi:10.1145/3372297.3417261.

6 Orestis Alpos, Ignacio Amores-Sesar, Christian Cachin, and Michelle Yeo. Eating sandwiches: Modular and lightweight elimination of transaction reordering attacks. *CoRR*, abs/2307.02954, 2023. arXiv:2307.02954, doi:10.48550/ARXIV.2307.02954.

7 Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *AFT*, pages 80–91. ACM, 2020. doi:10.1145/3419614.3423251.

8 Robert J. Aumann. Acceptable points in general cooperative n-person games. In Albert William Tucker and Robert Duncan Luce, editors, *Contributions to the Theory of Games (AM-40), Volume IV*, pages 287–324. Princeton University Press, Princeton, 1959. doi:10.1515/9781400882168-018.

9 Leemon Baird and Atul Luykx. The hashgraph protocol: Efficient asynchronous BFT for high-throughput distributed ledgers. In *COINS*, pages 1–7. IEEE, 2020. doi:10.1109/COINS49042.2020.9191430.

10 Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. Sok: Mitigation of front-running in decentralized finance. *IACR Cryptol. ePrint Arch.*, page 1628, 2021. URL: https://eprint.iacr.org/2021/1628.

11 Carsten Baum, Bernardo David, and Rafael Dowsley. Insured MPC: efficient secure computation with financial penalties. In *Financial Cryptography*, volume 12059 of *Lecture Notes in Computer Science*, pages 404–420. Springer, 2020. doi:10.1007/978-3-030-51280-4_22.

12 Carsten Baum, Bernardo David, and Tore Kasper Frederiksen. P2DEX: privacy-preserving decentralized cryptocurrency exchange. In *ACNS (1)*, volume 12726 of *Lecture Notes in Computer Science*, pages 163–194. Springer, 2021. doi:10.1007/978-3-030-78372-3_7.

13 B. Douglas Bernheim, Bezalel Peleg, and Michael D Whinston. Coalition-proof nash equilibria I. Concepts. *Journal of Economic Theory*, 42(1):1–12, 1987. doi:10.1016/0022-0531(87)90099-8.

14 Manuel Blum. Coin flipping by telephone a protocol for solving impossible problems. *SIGACT News*, 15(1):23–27, jan 1983. doi:10.1145/1008908.1008911.

15 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 757–788. Springer, 2018. doi:10.1007/978-3-319-96884-1_25.

16 Mic Bowman, Debajyoti Das, Avradip Mandal, and Hart Montgomery. On elapsed time consensus protocols. In *INDOCRYPT*, volume 13143 of *Lecture Notes in Computer Science*, pages 559–583. Springer, 2021. doi:10.1007/978-3-030-92518-5_25.

17 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.

18 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001. doi:10.1007/3-540-44647-8_31.

19 Christian Cachin, Jovana Micic, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *Financial Cryptography*, volume 13411 of *Lecture Notes in Computer Science*, pages 316–333. Springer, 2022. doi:10.1007/978-3-031-18283-9_15.

20 Chainlink Labs. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. Whitepaper, 2021. URL: https://research.chain.link/whitepaper-v2.pdf.

21 Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Arash Pourdamghani. Probabilistic smart contracts: Secure randomness on the blockchain. In *IEEE ICBC*, pages 403–412. IEEE, 2019. doi:10.1109/BLOC.2019.8751326.

22 Kevin Choi, Arasu Arun, Nirvan Tyagi, and Joseph Bonneau. Bicorn: An optimistically efficient distributed randomness beacon. *IACR Cryptol. ePrint Arch.*, page 221, 2023. URL: https://eprint.iacr.org/2023/221.

23 Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *FOCS*, pages 383–395. IEEE Computer Society, 1985. doi:10.1109/SFCS.1985.64.

24 Bram Cohen and Krzysztof Pietrzak. The chia network blockchain. Whitepaper, 2019. URL: https://www.chia.net/wp-content/uploads/2022/07/ChiaGreenPaper.pdf.

25 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

26 Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *IEEE Symposium on Security and Privacy*, pages 910–927. IEEE, 2020. doi:10.1109/SP40000.2020.00040.

27 Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018. doi:10.1007/978-3-319-78375-8_3.

28 Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 1989. doi:10.1007/0-387-34805-0_28.

29 Yael Doweck and Ittay Eyal. Multi-party timed commitments. *CoRR*, abs/2005.04883, 2020. arXiv:2005.04883.

30 Sisi Duan, Michael K. Reiter, and Haibin Zhang. Secure causal atomic broadcast, revisited. In *DSN*, pages 61–72. IEEE Computer Society, 2017. doi:10.1109/DSN.2017.64.

31 Lioba Heimbach and Roger Wattenhofer. Eliminating sandwich attacks with the help of game theory. In *AsiaCCS*, pages 153–167. ACM, 2022. doi:10.1145/3488932.3517390.

32 Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. In *APKC@AsiaCCS*, pages 3–14. ACM, 2022. doi:10.1145/3494105.3526239.

33 Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. *IACR Cryptol. ePrint Arch.*, page 1465, 2021. URL: https://eprint.iacr.org/2021/1465.

34 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.

35 John Kelsey, Luís T. A. N. Brandão, Rene Peralta, and Harold Booth. Nistir 8213. a reference for randomness beacons: Format and protocol version 2, 2011.

36 Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 705–734. Springer, 2016. doi:10.1007/978-3-662-49896-5_25.

37 Gillat Kol and Moni Naor. Games for exchanging information. In *STOC*, pages 423–432. ACM, 2008. doi:10.1145/1374376.1374437.

38 Kshitij Kulkarni, Theo Diamandis, and Tarun Chitra. Towards a theory of maximal extractable value I: constant function market makers. *CoRR*, abs/2207.11835, 2022. arXiv:2207.11835, doi:10.48550/ARXIV.2207.11835.

39 Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *AFT*, pages 25–36. ACM, 2020. doi:10.1145/3419614.3423263.

40 Arjen K. Lenstra and Benjamin Wesolowski. Trustworthy public randomness with sloth, unicorn, and trx. *Int. J. Appl. Cryptogr.*, 3(4):330–343, 2017. doi:10.1504/IJACT.2017.10010315.

41 Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous MPC and its application to anonymous communication. In *CCS*, pages 887–903. ACM, 2019. doi:10.1145/3319535.3354238.

42 Mohsen Alambardar Meybodi, Amir Kafshdar Goharshady, Mohammad Reza Hooshmandasl, and Ali Shakiba. Optimal mining: Maximizing bitcoin miners' revenues from transaction fees. In *Blockchain*, pages 266–273. IEEE, 2022. doi:10.1109/BLOCKCHAIN55522.2022.00044.

43 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *CCS*, pages 31–42. ACM, 2016. doi:10.1145/2976749.2978399.

44 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. doi:10.1017/CBO9780511813603.

45 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, 2009. URL: http://bitcoin.org/bitcoin.pdf.

46 Martin J. Osborne and Ariel Rubinstein. *A course in game theory*. The MIT Press, Cambridge, USA, 1994. electronic edition.

47 Torben P. Pedersen. Cps, certificate practice statement. In *Encyclopedia of Cryptography and Security*. Springer, 2005. doi:10.1007/0-387-23483-7_83.

48 Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *IEEE Symposium on Security and Privacy*, pages 198–214. IEEE, 2022. doi:10.1109/SP46214.2022.9833734.

49 Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Trans. Program. Lang. Syst.*, 16(3):986–1009, 1994. doi:10.1145/177492.177745.

50 R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology, USA, 1996.

51 Nikolaj I. Schwartzbach. Deposit schemes for incentivizing behavior in finite games of perfect information. *CoRR*, abs/2107.08748, 2021. arXiv:2107.08748.

52 Avishay Yanai. Blinderswap: MEV meets MPC. https://www.youtube.com/watch?v=KQ4xK79YkFE&ab_channel=IC3InitiativeforCryptocurrenciesandContracts, 2021. Accessed 03/08/23.

53 Haoqian Zhang, Louis-Henri Merino, Vero Estrada-Galiñanes, and Bryan Ford. F3B: A low-latency commit-and-reveal architecture to mitigate blockchain front-running. *CoRR*, abs/2205.08529, 2022. doi:10.48550/ARXIV.2205.08529.

Improved Distributed Algorithms for Random Colorings

Charlie Carlson
Department of Computer Science, University of California, Santa Barbara, CA, USA

Daniel Frishberg
Department of Computer Science and Software Engineering, California Polytechnic State University, San Luis Obispo, CA, USA

Eric Vigoda
Department of Computer Science, University of California, Santa Barbara, CA, USA

Abstract

Markov Chain Monte Carlo (MCMC) algorithms are a widely-used algorithmic tool for sampling from high-dimensional distributions, a notable example is the equilibirum distribution of graphical models. The Glauber dynamics, also known as the Gibbs sampler, is the simplest example of an MCMC algorithm; the transitions of the chain update the configuration at a randomly chosen coordinate at each step. Several works have studied distributed versions of the Glauber dynamics and we extend these efforts to a more general family of Markov chains. An important combinatorial problem in the study of MCMC algorithms is random colorings. Given a graph G of maximum degree Δ and an integer $k \geq \Delta + 1$, the goal is to generate a random proper vertex k-coloring of G.

Jerrum (1995) proved that the Glauber dynamics has $O(n \log n)$ mixing time when $k > 2\Delta$. Fischer and Ghaffari (2018), and independently Feng, Hayes, and Yin (2018), presented a parallel and distributed version of the Glauber dynamics which converges in $O(\log n)$ rounds for $k > (2+\varepsilon)\Delta$ for any $\varepsilon > 0$. We improve this result to $k > (11/6 - \delta)\Delta$ for a fixed $\delta > 0$. This matches the state of the art for randomly sampling colorings of general graphs in the sequential setting. Whereas previous works focused on distributed variants of the Glauber dynamics, our work presents a parallel and distributed version of the more general flip dynamics presented by Vigoda (2000) (and refined by Chen, Delcourt, Moitra, Perarnau, and Postle (2019)), which recolors local maximal two-colored components in each step.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Graph algorithms analysis; Theory of computation → Random walks and Markov chains

Keywords and phrases Distributed Graph Algorithms, Local Algorithms, Coloring, Glauber Dynamics, Sampling, Markov Chains

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.13

Related Version *Full Version*: https://doi.org/10.48550/arXiv.2309.07859

Funding *Eric Vigoda*: Research supported in part by NSF grant CCF-2147094.

1 Introduction

This paper presents parallel and distributed algorithms for sampling from high-dimensional distributions. An important application is sampling from the equilibrium distribution of a graphical model. The equilibrium distribution is often known as the Gibbs or Boltzmann distribution, and efficient sampling from the Gibbs/Boltzmann distribution is a key step for Bayesian inference [24, 28].

Our focus is algorithms in the LOCAL model for the k-colorings problem. The k-colorings problem is a graphical model of particular combinatorial interest and has played an important role in the development of algorithmic sampling techniques with provable guarantees. The LOCAL model is a standard model of distributed computation due to Linial [26].

In the LOCAL model, the input to a problem is generally a graph $G = (V, E)$. Each vertex is identified with a processor and is assigned a unique identifier. In each *round* of an algorithm, each vertex is allowed to send an unbounded amount of information (a *message*) to each of its neighbors, and may perform an unbounded amount of computation locally.

For an input graph $G = (V, E)$ and integer $k \geq 2$, let Ω denote the proper (vertex) k-colorings of G, namely $\Omega = \{\sigma : V \to \{1, \ldots, k\} : \text{ for all } (v, w) \in E, \sigma(v) \neq \sigma(w)\}$ is the collection of assignments of k colors to the vertices so that neighboring vertices receive different colors. The associated Gibbs distribution μ is the uniform distribution over Ω, the space of proper k-colorings.

Under mild conditions on G (e.g., triangle-free [3]), the number of k-colorings is exponentially large, i.e., $|\Omega| = \exp(\Omega(n))$. Nevertheless, our goal is to sample from μ, the uniform distribution over this exponentially large set, in time $\text{poly}(n)$, and ideally in time $O(n \log n)$. Furthermore, in the distributed setting our goal is to generate samples ideally in time $O(\log n)$.

A common technique for sampling from the Gibbs distribution in a wide range of scientific fields is the *Markov Chain Monte Carlo (MCMC)* method. The simplest example of an MCMC algorithm is the Glauber dynamics, also known as the Gibbs sampler.

Consider an input graph $G = (V, E)$ with maximum degree Δ, and $k \geq \Delta + 2$. The Glauber dynamics updates the color of a randomly chosen vertex in each step. In particular, from a coloring $X_t \in \Omega$ at time t, the transitions $X_t \to X_{t+1}$ of the Glauber dynamics work as follows. We choose a random vertex v uniformly at random from V, and a color c uniformly at random from the set of colors $\{1, \ldots, k\}$. If no neighbor of v has color c in the current coloring X_t, i.e., $c \notin X_t(N(v))$ where $N(v)$ are the neighbors of vertex v, then we recolor v as $X_{t+1}(v) = c$ and otherwise we set $X_{t+1}(v) = X_t(v)$. For all other vertices $w \neq v$ we set $X_{t+1}(w) = X_t(w)$. This corresponds to the Metropolis version of the Glauber dynamics. Alternatively one can choose the color c uniformly from $\{1, \ldots, k\} \setminus X_t(N(v))$, which is the set of colors that do not appear in the neighborhood of v in X_t; this is the heat-bath version of the Glauber dynamics.

When $k \geq \Delta + 2$ then the Glauber dynamics is ergodic and the unique stationary distribution is uniform over Ω. The *mixing time* is the number of steps, from the worst initial state X_0, so that the chain is within total variation distance $\leq 1/4$ of the stationary distribution (see Section 2.2 for a more formal definition).

There are various attempts at running asynchronous versions of the Glauber dynamics in the distributed setting, namely HOGWILD! [35, 40], but there are few theoretical results and the resulting process is not guaranteed to have the correct asymptotic distribution [8, 7, 36]. There is also considerable work in constructing distributed sampling algorithms, including distributed versions of the Glauber dynamics [14, 13, 22, 12, 11, 27]; we discuss below the relevant results in our setting of the colorings problem. An important caveat about previous results is that they require a strong form of decay of correlations, such as the Dobrushin uniqueness condition, and our results hold in regions where Dobrushin's uniqueness condition does not hold.

In the sequential setting, a seminal work of Jerrum [21] proved $O(n \log n)$ mixing time of the Glauber dynamics whenever $k > 2\Delta$ where Δ is the maximum degree. Vigoda [37] presented an alternative dynamics which we will refer to as the flip dynamics and proved

$O(n \log n)$ mixing time of the flip dynamics when $k > \frac{11}{6}\Delta$. The flip dynamics is a generalization of the Glauber dynamics which "flips" maximal 2-colored components (clusters) in each step by interchanging the pair of colors on the chosen cluster; Vigoda's analysis chooses particular flip probabilities which depend on the size of the chosen cluster and do not flip any cluster larger than size six.

Vigoda's result was recently improved to $k > (\frac{11}{6} - \varepsilon_0)\Delta$ for some fixed $\varepsilon_0 \approx 10^{-5}$ by Chen, Delcourt, Moitra, Perarnau, and Postle [5]. This later result of $k > (\frac{11}{6} - \varepsilon_0)\Delta$ is the best known result for general graphs. There are various improvements (e.g., [9, 6]), however they all require particular girth or maximum degree assumptions; the girth is the length of the shortest cycle.

In the distributed setting, Feng, Sun and Yin [12] achieved $O(\Delta \log n)$ rounds in LOCAL model when $k > (2+\varepsilon)\Delta$ and $O(\log n)$ rounds when $k > (2+\sqrt{2})\Delta$. Fischer and Ghaffari [14], and independently, Feng, Hayes and Yin [11], presented a distributed algorithm which converges in $O(\log n)$ rounds for k-colorings on any graph of maximum degree Δ when $k > (2+\varepsilon)\Delta$ for any $\varepsilon > 0$. These results match Jerrum's result (in the sequential setting) for general graphs. We improve upon these works to match the current state of the art results in the sequential setting for general graphs for $k > (11/6)\Delta$.

We present the following improved result:

▶ **Theorem 1.** *For all $\varepsilon > 0$, all $\Delta \geq 2$, all $\delta > 0$, and any $k > (11/6 + \varepsilon)\Delta$, for any graph $G = (V, E)$ of maximum degree Δ, a random k-coloring within total variation distance $\leq \delta$ from uniform can be generated in $O(\log(n/\delta))$ rounds, where $n = |V|$.*

The above result is optimal as there is a matching $\Omega(\log(n/\delta))$ lower bound due to Feng, Sun, and Yin [12]. Moreover, combining our analysis with the refined analysis of Chen et al. [5] we obtain the following result.

▶ **Theorem 2.** *There exists $\varepsilon^* > 0$, for all $\Delta \geq 2$, all $\delta > 0$, and any $k > (11/6 - \varepsilon^*)\Delta$, for any graph of maximum degree Δ, a random k-coloring within total variation distance $\leq \delta$ from uniform can be generated in $O(\log(n/\delta))$ rounds.*

The Dobrushin uniqueness condition, which is a sufficient condition in several previous distributed sampling works, holds for colorings on general graphs of maximum degree Δ iff $k > 2\Delta$ [33]. Thus, our results hold beyond the Dobrushin uniqueness threshold, and thereby resolves an open problem of [14] who asked "whether efficient distributed algorithms intrinsically need to be stuck at Dobrushin's condition."

Our proof of fast convergence of our new distributed flip dynamics utilizes the path coupling framework of Bubley and Dyer [4], which is an important tool in the analysis of the mixing time for sequential Markov chains. In a coupling analysis path coupling allows one to only consider "neighboring pairs". In the special case of the Glauber dynamics, path coupling is related to Dobrushin's uniqueness condition but path coupling is a weaker condition (namely, Dobrushin's uniqueness condition implies path coupling). We believe our work raises the following intriguing open question. For any spin system, or equivalently any undirected graphical model, does the path coupling condition for a local (sequential) Markov chain imply the existence of an efficient distributed algorithm which converges in $O(\log n)$ steps?

1.1 Motivation

Designing a distributed algorithm for constructing a coloring is a seminal problem in the study of distributed algorithms [26, 29]. It is an important problem in the study of symmetry breaking and is useful in the design of networking algorithms [2, 34, 25, 26]. One of the

fundamental problems in this context that has received significant attention is minimizing the number of rounds required to construct a $(\Delta + 1)$-coloring in the LOCAL model; see Barenboim, Elkin, and Goldenberg [1] for a recent breakthrough, and see [15, 16] for more recent follow-up works.

Our focus is on generating a random coloring, in other words to generate a sample from the uniform distribution over all colorings, or more precisely, from a distribution that is arbitrarily close (in total variation distance) to the uniform distribution. More generally, our goal is to sample from the equilibrium distribution of a graphical model.

Graphical models are a fundamental tool in machine learning [28], and the associated sampling problem is important for associated learning, inference, and testing problems. A noteworthy example in the history of graphical models and in the importance of the associated sampling problem is the work on Restricted Boltzmann Machines (RBMs) of Hinton [18]. An RBM is an instance of the Ising model on a bipartite graph. The Ising model is a simpler variant of the random colorings problem in which we are sampling labelings of the vertices of a bipartite graph with only 2 colors where the labelings are weighted exponentially by the number of monochromatic edges; the generalization to $k > 2$ colors is the Potts model, and the zero-temperature (antiferromagnetic) Potts model is the random colorings problem that we study. The design of fast learning algorithms for RBMs was fundamental in the development of deep learning algorithms [19, 20, 30, 31, 32].

Given the proliferation of machine learning tasks on high-dimensional data, there is a clear need for distributed sampling algorithms for graphical models. For example, speeding up inference in *latent Dirichlet allocation* models via parallel and distributed Gibbs sampling [38, 23] and via the stochastic gradient sampler [39] has received attention in the machine learning community, as has the distributed problem of finding a k-coloring as a subroutine for Gibbs sampling [17].

Sampling colorings is a natural combinatorial problem to address particularly because of its importance in the study of sequential sampling algorithms. Jerrum's sampling algorithm [21] for $k > 2\Delta$ colors was a seminal work as it pioneered the use of the coupling method for sampling problems on graphical models. As mentioned earlier, Vigoda [37] improved Jerrum's result to $k > 11\Delta/6$ and this was the state of the art until the recent improvement to $k > (11/6 - \varepsilon)\Delta$ [5]. One of the major open problems in the area of sequential sampling is to obtain an efficient sampling scheme when $k > \Delta + 1$, see [6] for the most recent progress.

Our general question is whether efficient sequential sampling schemes yield efficient distributed sampling algorithms, by which we mean an $O(\log n)$ round algorithm in the LOCAL model. A distributed version of the Metropolis version of the Glauber dynamics for colorings was introduced in [14, 11] and was proved to be an efficient distributed sampling scheme when $k > (2 + \varepsilon)\Delta$ for all $\varepsilon > 0$. Our work goes beyond the single-site Glauber dynamics to designing efficient distributed sampling schemes for more general dynamics.

1.2 Technical Contribution

Recall that the Glauber dynamics updates a single vertex in each step. Several recent works present and analyze distributed versions of the Glauber dynamics (specifically, the Metropolis version) in various contexts [14, 11, 27, 12]. For more general MCMC algorithms which update larger regions than a vertex in each step, do efficient convergence results in the sequential setting for such Markov chains yield efficient distributed sampling algorithms?

A prime example to consider for this more general question is Vigoda's flip dynamics [37]. Attaining a distributed version of the flip dynamics is more challenging as we need to simultaneously recolor clusters of up to 6 vertices; here a cluster refers to a maximal 2-colored

component and the recoloring acts by interchanging the respective pair of colors on each cluster. Our first contribution is presenting a distributed version of Vigoda's flip dynamics. The challenge is to make a distributed version which is efficient but simple enough that we can still analyze it.

To parallelize the cluster recolorings, we need to ensure that no two overlapping clusters are simultaneously active, and that no two neighboring clusters that share colors are both active. On the other hand, we need to "activate" each cluster for potential recoloring with a sufficiently large probability to obtain a mixing time that is independent of the maximum degree, namely $O(\log n)$.

Our analysis of our distributed version of Vigoda's flip dynamics follows the high-level coupling presented in Vigoda's original work [37]. A coupling analysis of a Markov chain, considers two copies of the Markov chain (in this case the distributed flip dynamics), each with arbitrary starting states. Our aim is that there are "coupled transitions" for the two chains so that after $O(\log n)$ steps the two chains have coalesced in the same state with sufficiently large probability; by coupled transition we mean that the two chains can couple their transitions as long as when viewed in isolation, each is a faithful copy of the original Markov chain. The idea is that if we consider one of the chains to be in the stationary distribution, then we showed that after $O(\log n)$ steps our algorithm has likely reached the stationary distribution and hence the mixing time is $O(\log n)$.

There are several important technical challenges that arise when doing a coupling analysis in the distributed setting for the flip dynamics. First, we need to ensure that the clusters we flip (which means swap the pair of colors in a maximal 2-colored component) do not interfere with any other clusters we might flip by either overlapping, or by neighboring and containing a common color. Subsequently when we do try to couple a pair of flips in the two coupled chains, we need to consider the case that one of these two clusters is not flippable in only one chain due to one of these aforementioned conflicts (such as an overlapping cluster in only one of the chains).

Finally, similar to the original analysis of Vigoda [37], we use the path coupling framework [4] from which we only need to design and analyze a coupling for pairs of chains that differ at a single vertex, which we call v^*; in contrast, without path coupling we need to analyze pairs of chains that differ on an arbitrary number of vertices. However, the coupling analysis for a pair of chains X_t, Y_t that differ at this single vertex v^* is more complicated than in Vigoda's sequential setting. In Vigoda's original analysis, the only pertinent cluster flips in X_t or Y_t are those clusters that either include v^* or include a neighbor of v^*. In our analysis in the distributed setting, we also need to consider the effect from clusters that are distance exactly 2 away from v^*, where distance is measured by cluster adjacencies. These distance-2 clusters are identical sets of vertices in both chains X_t and Y_t but they may be flippable in only one of the chains (due to differing distance-1 clusters).

Our work suggests that a more general phenomenon is at play. We conjecture that, for any graphical model, a path coupling analysis for any local Markov chain in the sequential setting yields an efficient distributed sampling scheme. We believe our work will be an important step towards proving this general conjecture.

1.3 Paper Overview

In Section 3, we present a parallel and distributed version of Vigoda's flip dynamics. We analyze the mixing time of our distributed flip dynamics when $k > (11/6 + \varepsilon)\Delta$ for any $\varepsilon > 0$, thereby proving Theorem 1, in Sections 4 and 5. We use a coupling argument that builds upon the analysis in Vigoda [37]. Our analysis is more complicated than the original

argument of Vigoda due to clusters which appear in both coupled chains, but possibly being "flippable" in one chain but not the other chain due to differing conflicts with neighboring clusters, see Section 1.2 for a very high-level overview. In the appendix of the full version of this paper, we further utilize the linear programming (LP) framework and the refined metric on colorings presented in Chen et al. [5] to achieve the further improved result as stated in Theorem 2. This proof combines our proof approach for Theorem 1 with the more technical analysis of Chen et al. [5].

2 Preliminaries

Let $[k] = \{1, \dots, k\}$. For a graph $G = (V, E)$ let $i \sim j$ denote $(i, j) \in E$, and for $v \in V$, let $N(v) = \{w \in V : v \sim w\}$ denote the neighbors of a vertex v. For integer $k \geq 2$, let $\Omega^* = [k]^V$ denote the set of *k-labelings* and $\Omega = \{\sigma \in [k]^V : \text{for all } i \sim j, \sigma(i) \neq \sigma(j)\}$ denote the set of *k-colorings* of G. Throughout this paper, a coloring (or k-coloring) refers to a proper vertex k-coloring.

2.1 Clusters

For a coloring σ, a cluster S in σ is a maximal 2-colored component of *size at most 6*; this is formally defined in the following definition.

▶ **Definition 3.** *Let $G = (V, E)$ be a graph and $\sigma \in \Omega^*$. For a vertex $v \in V$ and color $c \in [k]$ let $S_\sigma(v, c)$ denote the set of vertices reachable from v by a $(\sigma(v), c)$ alternating path, i.e., a path of vertices $v = v_1, v_2, \dots, v_\ell \in V$ for some $\ell \geq 1$ that alternate between colors $\sigma(v)$ and c. When $|S_\sigma(v, c)| \leq 6$ then we refer to $S = S_\sigma(v, c)$ as a* cluster*. Let*

$$\mathcal{S}_\sigma = \bigcup_{v \in V, c \in [k]} \{S_\sigma(v, c) : |S_\sigma(v, c)| \leq 6\},$$

denote the collection of all clusters in σ of size at most 6, where the size of a cluster refers to the number of vertices in the cluster. The restriction to size at most 6 is due to the Markov chain used as in previous works [37, 5].

The key operation of our Markov chain is "flipping" clusters which we define now.

▶ **Definition 4.** *For a labeling $\sigma \in \Omega^*$, vertex $v \in V$, and color $c \in [k]$, the* flip *of cluster $S_\sigma(v, c)$ interchanges colors $\sigma(v)$ and c on the set $S_\sigma(v, c)$.*

Let σ' denote the resulting coloring after this flip of cluster $S_\sigma(v, c)$. Notice that if $\sigma \in \Omega$ then $\sigma' \in \Omega$, i.e., if it is a proper coloring before the flip, then after the flip it remains a proper coloring since the clusters are maximal 2-colored components. Hence for Vigoda's flip dynamics, if we start the flip dynamics at a proper coloring, i.e., $X_0 \in \Omega$, then we are guaranteed to stay at proper colorings, i.e., $X_t \in \Omega$ for all $t \geq 0$.

The subsequent flip dynamics defined in Section 3 is defined on the set Ω, which is the set of proper k-colorings, and for algorithmic purposes one only needs to consider proper colorings. The extension of the state space to Ω^*, which is the set of all labelings, is only needed in the proof for technical reasons pertaining to the use of the path coupling method [4], which we present in Section 2.3. The introduction of improper colorings in the coupling analysis arises in all related path coupling proofs for colorings [4, 37, 5], see Section 4.2 for further discussion of this technicality of introducing improper colorings in the proof.

Consider a coloring $\sigma \in \Omega$ and a vertex $v \in V$. For every color c which does not appear in the neighborhood of v, i.e., $c \notin \sigma(N(v))$ then the corresponding cluster is of size 1, i.e., $|S_\sigma(v, c)| = 1$ since $S_\sigma(v, c) = \{v\}$. Flips of these singleton clusters are exactly the transitions

of the Glauber dynamics. The flip dynamics of Vigoda [37] is a generalization of the Glauber dynamics in which clusters of size at most 6 are flipped with positive probability (depending on the size of the cluster). Note, for $c = \sigma(v)$ then we get a singleton cluster and the flip does not change the coloring, hence the flip dynamics has a non-zero self-loop probability and thus is aperiodic.

For clusters $S, T \in \mathcal{S}_\sigma$, we say S and T are neighboring clusters, which we denote as $S \sim T$, if there exists $v \in S$ and $w \in T$ where $v \sim w$.

2.2 Markov Chains

Consider a Markov chain (X_t) with state space Ω and transition matrix P and unique stationary distribution π. We say that the chain is *aperiodic* if $\gcd\{t \colon P^t(x,x) > 0\} = 1$ for all $x \in \Omega$ and *irreducible* if for all $x, y \in \Omega$, there exists a t such that $P^t(x,y) > 0$. Recall that if the chain is both aperiodic and irreducible, then it is *ergodic* and the chain has a unique *stationary distribution* π where: for all $x, y \in \Omega, \lim_{t\to\infty} P^t(x,y) = \pi(y)$. If P is symmetric, then π is the uniform distribution over Ω.

The *mixing time* is the number of steps, from the worst initial state X_0, until the chain is within total variation distance $\leq 1/4$ of the stationary distribution:

$$T_{\text{mix}} := \max_{x\in\Omega} \min\{t \mid \|P^t(\sigma,\cdot) - \pi\|_{\text{TV}} \leq 1/4\},$$

where d_{TV} is the *total variation distance*, $\|\mu - \omega\|_{\text{TV}} := \frac{1}{2}\sum_{x\in\Omega}|\mu(x) - \omega(x)|$. The choice of constant $1/4$ is somewhat arbitrary since, for any $\varepsilon > 0$, we can obtain total variation distance $\leq \varepsilon$ after $\leq \log(1/\varepsilon)T_{\text{mix}}$ steps.

2.3 Path Coupling

Consider an ergodic Markov chain $\mathcal{MC}$ with state space Ω and transition matrix P. A *coupling* for $\mathcal{MC}$ defines, for all pairs $X_t, Y_t \in \Omega$, a joint transition $(X_t, Y_t) \to (X_{t+1}, Y_{t+1})$ such that the individual transitions $(X_t \to X_{t+1})$ and $(Y_t \to Y_{t+1})$, when viewed in isolation from each other, act according to the transition matrix P. The goal is to find a coupling that minimizes the coupling time: $T_{\text{couple}} := \min\{t \mid \text{ for all } X_0, Y_0 \in \Omega, \mathbf{Pr}(X_t \neq Y_t \mid X_0, Y_0) \leq 1/4\}$. This implies that $T_{\text{mix}} \leq T_{\text{couple}}$.

To bound the coupling time and hence the mixing time, we use the use the *path coupling* method of Bubley and Dyer [4] which allows us to only consider a small subset of pairs of states. We will analyze the coupling with respect to the *Hamming distance* $H(\sigma,\tau) := \sum_{v\in V} \mathbf{1}(\sigma(v) \neq \tau(v))$. We present the more general form of path coupling in the appendix of the full version of this paper which allows more general metrics.

▶ **Theorem 5** ([4, 10]). *Consider an ergodic Markov chain on $\Omega^* = [k]^V$. Let $\beta > 0$. If for all pairs of states $X_t, Y_t \in \Omega^*$ where $H(X_t, Y_t) = 1$, there exists a coupling such that:*

$$\mathbf{E}(H(X_{t+1}, Y_{t+1}) \mid X_t, Y_t) \leq (1-\beta),$$

then the mixing time is bounded by $T_{mix} \leq O\left(\frac{\log(|V|)}{\beta}\right)$. Moreover, the mixing time within total variation distance $\leq \delta$, for any $\delta > 0$, in time $O(\log(|V|)/(\beta\delta))$.

3 Algorithm Description: Distributed Flip Dynamics

We begin by defining a sequential process and then show that this process can be implemented efficiently in a distributed manner.

We have the following parameters in our algorithm. Let $\alpha = \varepsilon/(5000k)$ where $k \geq (11/6+\varepsilon)\Delta$ for some $\varepsilon > 0$. The parameter α will be used for the activation probability of a cluster. In the full version of this paper when we strengthen the main result for $k < (11/6)\Delta$ we redefine α so that it depends on the distance of k below $(11/6)\Delta$.

Let $1 \geq f_i \geq 0$ for all $i \geq 1$ be a sequence of "flip" probabilities that contain the following key properties: $f_1 = 1$, $f_i \geq f_{i+1}$ for all i, and $f_i = 0$ for all $i \geq 7$. The following process is well-defined for any choice of flip probabilities with these properties. To prove Theorems 1 and 2 we will choose slightly different flip probabilities. In particular, to prove the slightly weaker result (Theorem 1) in Section 4 we will choose flip probabilities as in [37], and then to get the refined result (Theorem 2) in the appendix of the full version of this paper we will use the setting in [5].

We now define the Markov chain $\mathcal{MC}_{\text{flip}}$ with state space Ω. For a coloring $X_t \in \Omega$, the transitions $X_t \to X_{t+1}$ of $\mathcal{MC}_{\text{flip}}$ are defined as follows:

1. Independently for each $S \in \mathcal{S}_\sigma$, cluster S is active with probability α.
2. A cluster $S = S_{X_t}(v, c)$ is flippable if the following hold:
   - (a) S is active;
   - (b) *Overlapping clusters:* There is no active $S' \neq S$ where $S \cap S' \neq \emptyset$;
   - (c) *Conflicting neighboring clusters:* For all active clusters $T = T_{X_t}(w, c')$ where $S \sim T$, $\{X_t(v), c\} \cap \{X_t(w), c'\} = \emptyset$.
3. Independently for each flippable cluster S, flip S with probability f_i where $i = |S|$.
4. Let X_{t+1} denote the resulting coloring.

Notice that step 2c is saying that for a pair of active and neighboring clusters S and T, the pair of colors defining cluster S are disjoint from the pair of colors defining cluster T.

▶ **Lemma 6.** *The Markov chain $\mathcal{MC}_{\text{flip}}$ is ergodic and symmetric and hence the unique stationary distribution is the uniform distribution over Ω.*

Proof. Observe that with positive probability, no cluster is active and $P(\sigma, \sigma) > 0$ for all $\sigma \in \Omega$. Thus, the Markov chain is aperiodic. For irreducibility, since $f_1 > 0$, the irreducibility of $\mathcal{MC}_{\text{flip}}$ follows from irreducibility of the Glauber dynamics which holds whenever $k \geq \Delta + 2$ (see, e.g., Jerrum [21]). Hence, the chain is ergodic. Moreover, the chain is symmetric, for $\sigma \in \Omega$, let σ' be the coloring obtained from σ after flipping clusters $S_\sigma(v_1, c_1), \ldots, S_\sigma(v_\ell, c_\ell)$ in one step of $\mathcal{MC}_{\text{flip}}$. Then, starting from σ' and flipping clusters $S_{\sigma'}(v_1, \sigma(v_1)), \ldots, S_{\sigma'}(v_\ell, \sigma(v_\ell))$ recovers σ. Since $\mathcal{MC}_{\text{flip}}$ is ergodic and symmetric then the uniform distribution is the unique stationary distribution. ◀

▶ **Lemma 7.** *Each step of the Markov chain $\mathcal{MC}_{\text{flip}}$ can be implemented in the LOCAL model in $O(1)$ rounds.*

Proof. We describe the steps of the algorithm and how to implement them in the LOCAL model. At a given time step t, denote the current coloring as $\sigma = X_t$.

1. For each vertex $v \in V$ and for each color $c \in [k]$, identify the cluster $S_\sigma(v, c)$. We accomplish this step by (i) sending a message indicating the index of v to each neighboring vertex w with $\sigma(w) = c$, (ii) passing this message, along with the index of w, to each neighbor x of w with $\sigma(x) = \sigma(v)$, and (iii) repeating this process for up to six rounds. After the six rounds, each vertex has received the identities of all other vertices in its six-hop neighborhood with which it might share a cluster, and thus can determine the clusters (and their sizes) to which it belongs. Moreover, any 2-colored components of size > 6 will be identified and discarded.

2. Fix an arbitrary ordering of the vertex set of G and for each cluster S, identify $\text{pres}(S)$, the lowest-index vertex $v \in S$. We can accomplish this step by letting each vertex $u \in S$ compare its own index to each of the indices of other vertices in S, which have been passed during step 1.
3. For each cluster S, activate S with probability α. More precisely, for each $v \in V$ and for each $c \in [k]$, if $v = \text{pres}(S_\sigma(v,c))$, activate $S_\sigma(v,c)$ by sending a message to every $u \in S_\sigma(v,c)$.
4. Detect conflicts:
   (a) *Overlapping clusters:* For all $v \in V$, if $S_\sigma(v,c), S_\sigma(v,c')$ are both active for some $c \neq c'$, send messages to $\text{pres}(S_\sigma(v,c)), \text{pres}(S_\sigma(v,c'))$ to "deactivate" $S_\sigma(v,c), S_\sigma(v,c')$.
   (b) *Conflicting neighboring clusters:* For all $v \in V$, for every neighbor w of v, if there exist clusters $S_\sigma(v,c) \neq S_\sigma(w,c')$ such that $\{\sigma(v), c\} \cap \{\sigma(w), c'\} \neq \emptyset$ and if $S_\sigma(v,c)$ and $S_\sigma(w,c')$ are both active, deactivate $S_\sigma(v,c)$ and $S_\sigma(w,c)$ (by sending messages to $\text{pres}(S_\sigma(v,c))$ and $\text{pres}(S_\sigma(w,c))$).
5. For all $v \in V$, for all $c \in [k]$, if $S_\sigma(v,c)$ is still active and $v = \text{pres}(S_\sigma(v,c))$, flip $S_\sigma(v,c)$ with probability f_i, where $i = |S_\sigma(v,c)|$ (by sending a message to each $w \in S_\sigma(v,c)$ to change its color from c to $\sigma(v)$ or vice versa).

Since, in step 5, only $\text{pres}(S)$ is responsible for flipping S, the probability of a given cluster S being flipped, conditioned on S being active and having no active neighboring or overlapping cluster, is $f_{|S|}$.

Each of the above steps requires a constant number of rounds, proving the claim. Furthermore, the amount of computation performed locally at each vertex depends only (and polynomially) on the maximum degree of the graph and the number of colors. That is, not only is the number of rounds in the LOCAL model $O(1)$, but also the algorithm is efficient with respect to the local computation performed in each round. ◀

4 Analysis of Distributed Flip Dynamics

Here we prove our main result Theorem 1, namely fast convergence of the distributed flip dynamics when $k > (11/6 + \varepsilon)\Delta$ for any $\varepsilon > 0$. Hence, fix $\varepsilon > 0$ and $k > (11/6 + \varepsilon)\Delta$. Our specific choice of flip probabilities for this section and for Section 5 are the following:

$$f_1 = 1,\ f_2 = 13/42,\ f_3 = 1/6,\ f_4 = 2/21,\ f_5 = 1/21,\ f_6 = 1/84. \tag{1}$$

These parameters match the original paper of Vigoda [37]; there are other parameter choices for which the analysis works, e.g., see [5], in fact, we will utilize these alternative parameters in the appendix of the full version of this paper.

4.1 Overview

We will analyze the mixing time of the chain $\mathcal{MC}_{\text{flip}}$ using path coupling. Consider a pair of colorings X_t, Y_t which differ at exactly one vertex and let v^* denote the disagreement, i.e., $X_t(v^*) \neq Y_t(v^*)$ and for all $w \neq v^*$, $X_t(w) = Y_t(w)$. Our coupling is the identity coupling for all clusters that are the same in both chains, i.e., for all clusters S where $S = S_{X_t}(w,c) = S_{Y_t}(w,c)$ for some $w \in V, c \in [k]$, we use the identity coupling for the activation probability. By the identity coupling for the activation probability we mean that with probability α the cluster S is active in both chains, and with probability $1 - \alpha$ it is inactive in both chains. Moreover, if the cluster S is flippable in both chains then we also use

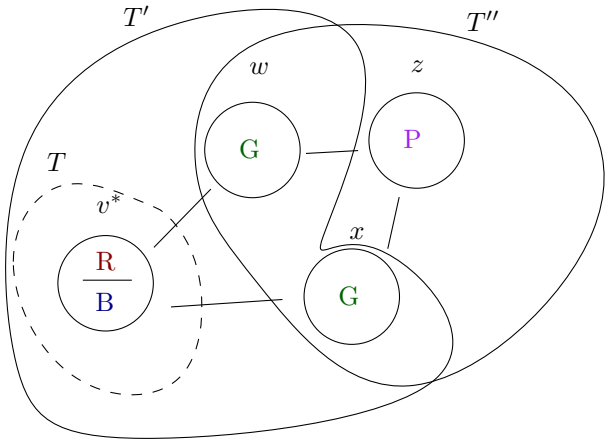


Figure 1 The vertex v has $X_t(v) = R$ and $Y_t(v) = B$. Here, $T = S_\sigma(v, R) = S_\tau(v, B)$, where $\sigma = X_t$ and $\tau = Y_t$. By Definition 8, $\text{dist}(v^*, T) = 0, \text{dist}(v^*, T') = \text{dist}(v^*, T'') = 1$.

the identity coupling for the flip probability, which means that if both clusters are flippable then with probability α we flip the cluster in both chains and with probability $1 - \alpha$ we flip the cluster in neither of the chains.

We will define the distance of cluster T from the disagree vertex v^* based on the shortest path via neighboring clusters.

▶ **Definition 8.** *For a coloring $\sigma \in \{X_t, Y_t\}$, and a cluster $T \in \mathcal{S}_\sigma$, we define* $\text{dist}(v^*, T)$ *inductively as follows. If $T = \{v^*\}$ then let* $\text{dist}_\sigma(v^*, T) = 0$. *In general, let*

$$\text{dist}_\sigma(v^*, T) = \min\{i : \textit{there exists } S \in \mathcal{S}_\sigma \textit{ where } S \sim T, \text{dist}_\sigma(v^*, S) = i - 1\}.$$

▶ Remark 9. Note, this notion of distance is equivalent to the shortest path distance from the singleton cluster $\{v^*\}$ in the *cluster graph*; the cluster graph is the graph on all clusters in coloring σ where clusters S and T are adjacent if $S \sim T$. Distance 0 clusters are the singleton sets $\{v^*\}$ for every color which does not appear in the neighborhood of v^*. Distance 1 clusters are those that contain a neighbor of v^* (regardless of whether they also contain v^*).

Any clusters T where no vertex in T is adjacent to v^* are identical in the two chains, and thus, for every $i \geq 2$:

$$T \in \mathcal{S}_{X_t}, \text{dist}_{X_t}(v^*, T) = i \iff T \in \mathcal{S}_{Y_t}, \text{dist}_{Y_t}(v^*, T) = i.$$

Similarly, the only clusters T which "disagree" in the sense that they appear in only one chain then T is at distance 1 from v^*; more formally, if $T \in \mathcal{S}_{X_t} \setminus \mathcal{S}_{Y_t}$, then $\text{dist}_{X_t}(v^*, T) = 1$, and if $T \in \mathcal{S}_{Y_t} \setminus \mathcal{S}_{X_t}$, then $\text{dist}_{Y_t}(v^*, T) = 1$. We use $\text{dist}(v^*, T)$ when the distances are equal, i.e., $\text{dist}_{X_t}(v^*, T) = \text{dist}_{Y_t}(v^*, T)$.

For such clusters T where $\text{dist}(v^*, T) \geq 2$ we use the identity coupling for the activation probability in X_t and Y_t, and thus the cluster T is active in both chains or in neither chain. It follows that for clusters T' with $\text{dist}(v^*, T') \geq 3$ then the cluster is flippable in both chains or in neither chain, as their neighboring active clusters are identical in the two chains. Therefore, we can use the identity coupling for the flip probability of this cluster T if the cluster is flippable, and such clusters are flipped in both chains or neither chain; this leads to the following observation.

▶ **Observation 10.** *For any cluster T' where* $\text{dist}(v^*, T') \geq 3$, $X_{t+1}(T') = Y_{t+1}(T')$.

For clusters T where $\text{dist}_G(T, v^*) = 2$, it can occur that T is flippable in only one of the chains (due to a neighboring cluster at distance 1 that occurs in only one of the chains). Hence, there is a probability that such clusters can be a new disagreement. The upcoming Lemma 11 proves that this occurs with an arbitrarily small constant probability.

The following lemma bounds the expected increase in Hamming distance from flips on clusters at distance exactly 2 from v^*.

▶ **Lemma 11.**

$$\sum_{T:\text{dist}(v^*,T)=2} |T|\mathbf{Pr}\left(X_{t+1}(T) \neq Y_{t+1}(T)\right) \leq \alpha\varepsilon\Delta/10,$$

where $k \geq (\frac{11}{6} + \varepsilon)\Delta$.

We will account for these potential disagreements at distance 2 via the clusters at distance 1. For a cluster T at distance 1 to occur in only one of the chains, the pair of colors defining T must include color $X_t(v^*)$ or color $Y_t(v^*)$.

Proof of Lemma 11. Let $\mathcal{S}_{X_t} \oplus \mathcal{S}_{Y_t} := (\mathcal{S}_{X_t} \setminus \mathcal{S}_{Y_t}) \cup (\mathcal{S}_{Y_t} \setminus \mathcal{S}_{X_t})$ denote the set of clusters that appear in one chain but not in the other chain. Consider a cluster $S \in \mathcal{S}_{X_t} \oplus \mathcal{S}_{Y_t}$. Note, all such S are at $\text{dist}(v^*, S) = 1$.

Let $c_X = X_t(v^*)$ and $c_Y = Y_t(v^*)$. These clusters $S \in \mathcal{S}_{X_t} \oplus \mathcal{S}_{Y_t}$ are either:

$$S_{X_t}(w, c_X), S_{X_t}(w, c_Y), S_{Y_t}(w, c_X), \text{ or } S_{Y_t}(w, c_Y),$$

for some neighbor $w \in N(v^*)$. Hence, there are $\leq 4\Delta$ such clusters $S \in \mathcal{S}_{X_t} \oplus \mathcal{S}_{Y_t}$.

Each such cluster S has size ≤ 6 and hence it has $\leq 6 \cdot 2\Delta$ neighboring clusters T that share a color with S. These clusters T are at distance $= 2$ from v^*. Note that if S and T are both active then T is not flippable in one of the chains, but it may be flippable in the other chain where S does not appear; hence, the chains X_{t+1} and Y_{t+1} potentially differ at T. This yields the following:

$$\sum_{T:\text{dist}(v^*,T)=2} |T|\mathbf{Pr}\left(X_{t+1}(T) \neq Y_{t+1}(T)\right) \leq 6 \times (4\Delta)(12\Delta)\alpha^2 = 288\Delta^2\alpha^2 \leq \varepsilon\alpha\Delta/10. \quad \blacktriangleleft$$

We now account for the "good moves" where the disagreement at v^* is removed. This occurs by Glauber updates at v^* where we update v^* to an available color, which is a color that does not appear in its neighborhood.

▶ **Definition 12.** *Denote the set of* available colors *for* v^* *in* X_t *as:*

$$A(v^*) = A_{X_t}(v^*) := \{c : c \notin X_t(N(v^*))\}.$$

Note, the sets $A_{X_t}(v^*) = A_{Y_t}(v^*)$ since v^* is the only disagreement at time t. Consider a color $c \in A(v^*)$. The clusters involving c to which v^* belongs satisfy $S_{X_t}(v^*, c) = S_{Y_t}(v^*, c) = \{v^*\}$ and hence the identity coupling is used for this cluster. Therefore, with probability α the cluster is active in both chains and if no active clusters overlap and no neighboring clusters have a common color then v^* is recolored to c.

We can now bound the probability of v^* agreeing at time $t+1$ in terms of the number of available colors for v^*.

▶ **Lemma 13.** $\mathbf{Pr}\left(X_{t+1}(v^*) = Y_{t+1}(v^*)\right) \geq |A(v^*)|\alpha(1 - \varepsilon/500)$.

Proof. For each color $c \in A(v^*)$ note $S_{X_t}(v^*, c) = S_{Y_t}(v^*, c) = \{v^*\}$. Hence, for $c \in A(v^*)$, let $S_c = S_{X_t}(v^*, c) = S_{Y_t}(v^*, c)$ denote this cluster of size 1 which appears in both chains. Since S_c appears in both chains we use the identity coupling for being active so that with probability α the cluster S_c is active in both chains, and with probability $1 - \alpha$ the cluster S_c is inactive in both chains. The cluster S_c may have different neighboring clusters in the two chains (which affects whether it is flippable) but if it is flippable in both chains then with probability $f_1 = 1$ we flip the cluster in both chains.

There are at most 2Δ neighboring clusters in each chain that share a color with one of the respective cluster, and there are $k - 1$ clusters (namely those at v^*) that overlap with these clusters. If none of the $2 \cdot 2\Delta$ neighboring clusters is active, and none of the $2(k-1)$ overlapping clusters is active in either chain, then we can flip $\{v^*\}$ in both chains. After this flip, v^* agrees in both chains, and hence we obtain:

$$\begin{aligned}\mathbf{Pr}\left(X_{t+1}(v^*) = Y_{t+1}(v^*)\right) &\geq |A(v^*)|\alpha(1-\alpha)^{4\Delta+k-1} \\ &\geq |A(v^*)|\alpha \exp(-\varepsilon(4\Delta + k - 1)/2500k) \\ &\geq |A(v^*)|\alpha \exp(-\varepsilon/500) \\ &\geq |A(v^*)|\alpha(1 - \varepsilon/500),\end{aligned}$$

where the second inequality uses the fact that $1 - x \geq \exp(-2x)$ for $x \leq 1/2$. ◀

The upcoming lemma captures the potential disagreements that arise from flipping clusters at distance one. The coupling on clusters containing v^* or neighboring v^* in at least one chain will be coupled based on the new color c.

▶ **Definition 14.** *For a color $c \in [k]$, let $N_c(v^*) = \{w \in N(v) : X_t(w) = c\} = \{w \in N(v) : Y_t(w) = c\}$ denote the neighbors of v^* with color c, and let $d_c(v^*) = |N_c(v^*)|$ denote the number of neighbors of v^* with color c at time t.*

Let $\mathcal{S}_{X_t}(c)$ denote the collection of clusters at distance 1 in X_t that involve color c:

$$\mathcal{S}_{X_t}(c) := \{S_{X_t}(w, X_t(v^*)) \mid w \in N_c(v^*)\} \cup \{S_{X_t}(w, Y_t(v^*)) \mid w \in N_c(v^*)\},$$

and similarly let $\mathcal{S}_{Y_t}(c)$ denote the corresponding collection for the coloring Y_t.

The sets $\mathcal{S}_{X_t}(c)$ and $\mathcal{S}_{Y_t}(c)$ are coupled with each other. We will specify the detailed coupling later, for now all that is needed is that these sets $\mathcal{S}_{X_t}(c)$ and $\mathcal{S}_{Y_t}(c)$ are coupled with each other. We can now state the key lemma bounding the increase in Hamming distance when we do a coupled update on these sets $\mathcal{S}_{X_t}(c), \mathcal{S}_{Y_t}(c)$.

In the following statement, recall, that for $\sigma, \tau \in \Omega$, $H(\sigma, \tau) = |\{v \in V : \sigma(v) \neq \tau(v)\}|$ is the Hamming distance.

▶ **Lemma 15.** *Let $c \in [k]$ where $d_c(v^*) > 0$. Recall that the flips of clusters in $\mathcal{S}_{X_t}(c)$ for $X_t \to X_{t+1}$ are coupled with clusters in $\mathcal{S}_{Y_t}(c)$ for $Y_t \to Y_{t+1}$. Let $\mathcal{F}_c$ denote the event that one of these coupled flips occurred in at least one of the chains. Then,*

$$\sum_{c \in [k]} \mathbf{E}\left(H(X_{t+1}, Y_{t+1})\mathbf{1}(\mathcal{F}_c)\right) \leq \sum_{c \in [k]} \left[1 + \alpha(1 + \varepsilon/5)\left(\frac{11}{6} d_c(v^*) - 1\right)\right] + \alpha\varepsilon\Delta/10.$$

The proof of Lemma 15 is deferred to Section 5. Note the error term $\alpha\varepsilon\Delta/10$ is coming from Lemma 11. Combining the above lemmas we can prove the main result (this is the slightly weaker version for $k \geq (11/6 + \varepsilon)\Delta$ for any $\varepsilon > 0$).

4.2 Proof of Theorem 1

We can extend the definition of our Markov chain (see Section 3) to be over all labelings $\Omega^* = [k]^V$ instead of just proper colorings Ω. This is necessary to apply path coupling Theorem 5. An identical approach is used in both [37] and [5]. The reason for this extension of the state space is the following. In the path coupling analysis we start with a pair of chains X_t, Y_t that differ at a single vertex v^*. In the coupling analysis we may introduce a new disagreement at time $t+1$ at a neighbor $w \in N(v^*)$ where the pair of disagreements at time $t+1$ are colored as $X_{t+1}(w) = Y_{t+1}(v^*)$, $Y_{t+1}(w) = X_{t+1}(v^*)$ and $X_{t+1}(v^*) \neq Y_{t+1}(v^*)$. Hence, the Hamming distance between X_{t+1} and Y_{t+1} is two but the number of Glauber dynamics steps (or cluster flips) to go from X_{t+1} to Y_{t+1} is three, and therefore in the path coupling analysis this new disagreement at w increases the distance by two (even though the Hamming distance increases by just one). This issue is resolved by extending the space to labelings Ω, subsequently there is an intermediate improper coloring so that X_{t+1} and Y_{t+1} are distance two apart.

The definition of the Markov chain described in Section 3 is identical, we simply extend the state space. A set $S_\sigma(v, c)$ is still defined as the set of vertices reachable from v by a $(\sigma(v), c)$ alternating path. And hence the notion of a cluster is still the same as before. Note, that while the chain restricted to proper colorings is symmetric, this is not necessarily true for improper colorings. All of the bounds stated in Section 4 hold for possibly improper colorings $X_t, Y_t \in \Omega^*$.

Consider a labeling $X_0 \in \Omega^* \setminus \Omega$; note, X_0 is not a proper coloring since $X_0 \notin \Omega$. For $k \geq \Delta + 2$, there is a sequence of transitions with non-zero probability (e.g., a sequence of Glauber moves as in the proof of irreducibility) so that it reaches a proper coloring, i.e., $X_t \in \Omega$ for some $t \geq 0$. Moreover, for any proper coloring $X_t \in \Omega$ then it stays on proper colorings, i.e., $X_s \in \Omega$ for all $s \geq t$, as the process does not introduce improper colorings. Therefore, states in Ω are the only ones which have positive probability in the stationary distribution, and hence the stationary distribution of the chain is uniform over the set of proper colorings Ω, even though the state space is all labelings Ω^*.

If the initial state is restricted to Ω, i.e., X_0 is a proper coloring, then the chain is identical to the process defined in Section 2.2. Furthermore, since the mixing time is defined from the worst initial state then a mixing time upper bound for the chain defined on Ω^* implies the same bound on the mixing time for the chain from Section 2.2 defined only on Ω.

We now have all the tools necessary to prove Theorem 1.

Proof of Theorem 1. First consider the available colors for v^*. Note that, since there is an extra available color for every time a color repeats in $N(v^*)$, we have

$$|A(v^*)| \geq k - d(v^*) + \sum_{c \in [k]: d_c(v^*) \geq 2} (d_c(v^*) - 1), \tag{2}$$

where $d(v^*) = \sum_{c \in [k]} d_c(v^*) = |N(v^*)|$ is the degree of v^*.

Now by combining Lemmas 11, 13, and 15 we can complete the proof of the theorem:

$$\begin{aligned}
&\mathbf{E}\left(H(X_{t+1}, Y_{t+1}) \mid X_t, Y_t\right) \\
&\quad \leq 1 - \mathbf{Pr}\left(X_{t+1}(v^*) = Y_{t+1}(v^*)\right) + \sum_{c: d_c(v^*) > 0} \left(\mathbf{E}\left(H(X_{t+1}, Y_{t+1}) \mathbf{1}(\mathcal{F}_c)\right) - 1\right) \\
&\qquad + \sum_{T: \mathrm{dist}(v^*, T) = 2} |T| \mathbf{Pr}\left(X_{t+1}(T) \neq Y_{t+1}(T)\right) \\
&\quad \leq 1 - \alpha |A(v^*)|(1 - \varepsilon/5) + \alpha(1 + \varepsilon/5) \sum_{c \in [k]: d_c(v^*) > 0} \left(\frac{11}{6} d_c(v^*) - 1\right) + \alpha \varepsilon \Delta / 5
\end{aligned} \tag{3}$$

$$
\begin{aligned}
&= 1 - \alpha|A(v^*)|(1-\varepsilon/5) + \alpha(1-\varepsilon/5) \sum_{c\in[k]:d_c(v^*)>0} \left(\frac{11}{6}d_c(v^*) - 1\right) \\
&\quad + \alpha(2\varepsilon/5) \sum_{c\in[k]:d_c(v^*)>0} \left(\frac{11}{6}d_c(v^*) - 1\right) + \alpha\varepsilon\Delta/5 \\
&\le 1 - \alpha|A(v^*)|(1-\varepsilon/5) + \alpha(1-\varepsilon/5) \sum_{c\in[k]:d_c(v^*)>0} \left(\frac{11}{6}d_c(v^*) - 1\right) \\
&\quad + \alpha\varepsilon\Delta(22/30) + \alpha\varepsilon\Delta/5
\end{aligned}
$$

where Equation (3) follows from Lemmas 11, 13, and 15. Then using Equation (2) we get

$$
\begin{aligned}
\mathbf{E}\left(H(X_{t+1}, Y_{t+1}) \mid X_t, Y_t\right) &\le 1 - \alpha(1-\varepsilon/5)\left[k - \frac{11}{6}d(v^*)\right] + \alpha\varepsilon\Delta(28/30) && (4)\\
&\le 1 - \alpha(1-\varepsilon/5)\varepsilon\Delta + \alpha\varepsilon\Delta(28/30) && (5)\\
&\le 1 - \alpha\varepsilon\Delta(29/30) + \alpha\varepsilon\Delta(28/30) && (6)\\
&\le 1 - \varepsilon^2/60000,
\end{aligned}
$$

where Equation (5) uses that $k \ge (1+\varepsilon)\frac{11}{6}\Delta$ and Equation (6) uses that $(1-\varepsilon/5) \ge 29/30$ when $\varepsilon \le 1/6$. Note, the case when $\varepsilon > 1/6$ and $k > 2\Delta$ is handled by [14, 11] or can be handled in our analysis by setting α in terms of $1/\Delta$ instead of $1/k$. Finally, applying the path coupling Theorem 5 we obtain mixing time $O(\log n)$. Moreover, we obtain mixing time within total variation distance $\le \delta$, for any $\delta > 0$, in time $O(\log(n/\delta))$. ◀

5 Coupling Analysis for Neighboring Clusters

We now prove Lemma 15. Before delving into the proof we state several key properties of the settings for the flip probabilities in Equation (1):

1. For all integer $i, j \ge 1$, $i(f_i - f_{i+1}) + (j-1)(f_j - f_{j+1}) \le 5/6$.
2. For all integer $i \ge 1$, $2(i-1)f_i + f_{2i+1} \le 2/3$.

Fix a color $c \in [k]$ where $d_c(v^*) > 0$; we will consider two cases: $d_c(v^*) = 1$ or $d_c(v^*) \ge 2$.

5.1 Flippable Difference

▶ **Lemma 16.** *For any cluster C,* $\mathbf{Pr}\left(C \text{ is active and not flippable}\right) \le \alpha\varepsilon/250$.

Proof. The cluster C is active with probability α. Assuming C is active, there are two ways that C is not flippable, either (i) an overlapping cluster, or (ii) a neighboring cluster that shares a color with C. For case (i), since $|C| \le 6$ and each vertex is in k clusters, then the probability of a cluster that overlaps C also being active is $\le \alpha 6k\alpha$. For case (ii), there are $\le 6\Delta$ neighboring vertices, each has ≤ 2 clusters that share a color, and hence the probability of case (ii) is $\le \alpha 12\Delta\alpha$. Combining the above calculations we have the following:

$$
\mathbf{Pr}\left(C \text{ is active and not flippable}\right) \le \alpha(6k\alpha + 12\Delta\alpha) \le \alpha\frac{18k\varepsilon}{5000k} < \alpha\varepsilon/250. \qquad ◀
$$

5.2 Color Appears Once

Suppose $d_c(v^*) = 1$. Let $w \in N(v^*)$ be the unique neighbor where $X_t(w) = Y_t(w) = c$, and let $R := X_t(v^*)$ and $B := Y_t(v^*)$. We are coupling the clusters in the set $\mathcal{S}_{X_t}(c)$ with $\mathcal{S}_{Y_t}(c)$, and since $d_c(v^*) = 1$ these sets are the following:

$\mathcal{S}_{X_t}(c) = \{S_{X_t}(w,R), S_{X_t}(w,B)\}$ and $\mathcal{S}_{Y_t}(c) = \{S_{Y_t}(w,R), S_{Y_t}(w,B)\}$.

Observe $S_{X_t}(w,R) = S_{Y_t}(w,R) \cup \{v^*\}$ and $S_{Y_t}(w,B) = S_{X_t}(w,B) \cup \{v^*\}$. Let $i := |S_{Y_t}(w,R)|$ (hence, $|S_{X_t}(w,B)| = i+1$), and let $j := |S_{X_t}(w,B)|$ ($|S_{Y_t}(w,B)| = j+1$). Note, $i, j \geq 1$.

We couple the clusters in the following manner. With probability α, cluster $S_{X_t}(w,R)$ is active in X_t and $S_{Y_t}(w,R)$ is active in Y_t, while with probability $1-\alpha$ both of these clusters are inactive . Similarly, with probability α then both: cluster $S_{X_t}(w,B)$ is active in X_t and $S_{Y_t}(w,B)$ is active in Y_t.

Suppose that $S_{X_t}(w,R)$ and $S_{Y_t}(w,R)$ are both flippable. In this case we maximize the probability that we flip both clusters. Since $f_i \geq f_{i+1}$ then with probability f_{i+1} we flip both clusters $S_{X_t}(w,R)$ and $S_{Y_t}(w,R)$, assuming they were both flippable. Similarly, with probability f_{j+1} we flip both clusters $S_{X_t}(w,B)$ and $S_{Y_t}(w,B)$, assuming they were both flippable. Note in both of these cases where we flip both $S_{X_t}(w,R)$ and $S_{Y_t}(w,R)$ or we flip both $S_{X_t}(w,B)$ and $S_{Y_t}(w,B)$, then the Hamming distance does not change as the chains only differ at v^* after the coupled update.

Suppose that all 4 clusters were flippable. (Recall an active cluster S is flippable if there is no overlapping active cluster and no neighboring active cluster which shares one of the two colors with S.). Then after the above coupling of $S_{X_t}(w,R)$ with $S_{Y_t}(w,R)$, and $S_{X_t}(w,B)$ with $S_{Y_t}(w,B)$, there remains probability $f_j - f_{j+1}$ to flip $S_{X_t}(w,B)$, and probability $f_i - f_{i+1}$ to flip $S_{Y_t}(w,R)$. We maximally couple these remaining flips and hence with probability $\min\{f_i - f_{i+1}, f_j - f_{j+1}\}$ we couple the flips of clusters $S_{X_t}(w,B)$ and $S_{Y_t}(w,R)$. Note in this case where we flip both $S_{X_t}(w,B)$ and $S_{Y_t}(w,R)$ then the Hamming distance increases by $\leq (i+j-1)$ since $S_{X_t}(w,B) \cap S_{Y_t}(w,R) \supseteq \{w\}$.

In the above coupling, we considered 3 coupled flips of cluster pairs $S_{X_t}(w,R), S_{Y_t}(w,R)$; $S_{X_t}(w,B), S_{Y_t}(w,B)$; and $S_{X_t}(w,B), S_{Y_t}(w,R)$. For each pair, it may occur that one of these clusters is flippable and the other is not flippable (due to a neighboring or overlapping cluster also being active). In that case we flip the flippable cluster by itself. In which case, the Hamming distance increases by at most 6 since the cluster is of size at most 6. By Lemma 16 the probability of this occurring for a specific cluster is at most $\alpha\varepsilon/250$, and since there are 3 pairs we have the effect is at most $36\alpha\varepsilon/250 = \alpha\varepsilon/5$.

Let us assume without loss of generality that $i \leq j$ and hence $f_i - f_{i+1} \geq f_j - f_{j+1}$. Now we can simplify and summarize the effect of the above coupled flips that change the Hamming distance. Since the clusters are active with probability α, with probability $\leq \alpha(f_j - f_{j+1})$ we flip $S_{X_t}(w,B)$ and $S_{Y_t}(w,R)$ and then the Hamming distance increases by $\leq (i+j-1)$. Moreover, with probability $f_i - f_{i+1} - (f_j - f_{j+1})$ we flip $S_{X_t}(w,B)$ by itself and the Hamming distance increases by i. Therefore, we have the following:

$$\begin{aligned}
&\mathbf{E}\left(H(X_{t+1}, Y_{t+1})\mathbf{1}(\mathcal{F}_c)\right) \\
&\quad \leq 1 + \alpha(i+j-1)(f_j - f_{j+1}) + i((f_i - f_{i+1}) - (f_j - f_{j+1})) + \alpha\varepsilon/5 \\
&\quad = 1 + \alpha i(f_i - f_{i+1}) + (j-1)(f_j - f_{j+1}) + \alpha\varepsilon/5 \\
&\quad \leq 1 + \alpha\left(\frac{5}{6} + \varepsilon/5\right)
\end{aligned}$$

where the last inequality follows by Property 1.

5.3 Color Appears More Than Once

The analysis of the case when the color appears more than once, i.e., $d_c(v^*) > 1$, follows the same general approach as in Section 5.2 for the case $d_c(v^*) = 1$. In particular, we use the same high-level coupling as used by Vigoda [37] but in addition we use Lemma 16 to bound the probability that a cluster is flippable in one chain and the coupled cluster is not flippable in the other chain. We refer the reader to the appendix of the full version of this paper for details.

6 Proof of Theorem 2: Mixing below 11/6

Sections 4 and 5 present the proof of Theorem 1 which establishes $O(\log n)$ mixing time of the distributed flip dynamics when $k > (11/6 + \varepsilon)\Delta$ for all $\varepsilon > 0$. The improved result for $k > (11/6 - \varepsilon^*)\Delta$ for a fixed $\varepsilon^* > 0$ as stated in Theorem 2 is proved in the appendix of the full version of this paper.

The proof of Theorem 2 uses the new metric introduced in [5], which is a weighted Hamming distance. In particular, in [5] they identify the configurations on the local neighborhood of the disagree vertex $v*$ for which the coupling analysis is tight, these are referred to as extremal configurations. Hence, for a pair of configurations X_t, Y_t which differ at a single vertex v^*, let γ denote the fraction of neighbors of v^* in non-extremal configurations. Then, [5] defines a new weighted Hamming distance as $\mathcal{H}(X_t, Y_t) = 1 - \gamma\eta$ for an appropriately defined small constant $\eta > 0$.

Using this new weighting, [5] proves rapid mixing of the flip dynamics in the sequential setting for $k > (11/6 - \varepsilon^*)\Delta$. The challenge in their analysis is that one has to consider the effect of coupled flips which do not change the Hamming distance but simply change whether some neighbors of v^* are in extremal configurations.

To obtain Theorem 2 we combine the approaches of [5] with our analysis in Sections 4 and 5 of the effect of the distributed synchronization. However the analysis becomes considerably more complicated than in [5] because multiple clusters in the neighborhood of v^* can flip in a single step, this leads to many new cases where the new weighted Hamming distance can change. The detailed analysis is contained in the appendix of the full version of this paper.

References

1 Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\Delta + 1)$-coloring and applications. *Journal of the ACM*, 69(1):5:1–5:26, 2021. doi:10.1145/3486625.

2 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3), 2016. doi:10.1145/2903137.

3 Anton Bernshteyn, Tyler Brazelton, Ruijia Cao, and Akum Kang. Counting colorings of triangle-free graphs. *Journal of Combinatorial Theory, Series B*, 161:86–108, 2023. doi:10.1016/J.JCTB.2023.02.004.

4 Russ Bubley and Martin E. Dyer. Path coupling: a technique for proving rapid mixing in Markov chains. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 223–231, 1997. doi:10.1109/SFCS.1997.646111.

5 Sitan Chen, Michelle Delcourt, Ankur Moitra, Guillem Perarnau, and Luke Postle. Improved bounds for randomly sampling colorings via linear programming. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2216–2234, 2019. doi:10.1137/1.9781611975482.134.

6 Zongchen Chen, Kuikui Liu, Nitya Mani, and Ankur Moitra. Strong spatial mixing for colorings on trees and its algorithmic applications. In *Proceedings of the 64th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2023.

7 Constantinos Daskalakis, Nishanth Dikkala, and Siddhartha Jayanti. Hogwild!-Gibbs can be panaccurate. In *Proceedings of the 31st Advances in Neural Information Processing Systems (NeurIPS)*, pages 32–41, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract.html.

8 Christopher De Sa, Kunle Olukotun, and Christopher Ré. Ensuring rapid mixing and low bias for asynchronous Gibbs sampling. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, pages 1567–1576, 2016. URL: http://proceedings.mlr.press/v48/sa16.html.

9 Martin Dyer, Alan Frieze, Thomas P. Hayes, and Eric Vigoda. Randomly coloring constant degree graphs. *Random Structures & Algorithms*, 43(2):181–200, 2013. doi:10.1002/RSA.20451.

10 Martin Dyer and Catherine Greenhill. Random walks on combinatorial objects. *Surveys in Combinatorics*, pages 101–136, 1999.

11 Weiming Feng, Thomas P. Hayes, and Yitong Yin. Distributed symmetry breaking in sampling (optimal distributed randomly coloring with fewer colors). *arXiv preprint*, 2018. arXiv:1802.06953.

12 Weiming Feng, Yuxin Sun, and Yitong Yin. What can be sampled locally? In *Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 121–130, 2017. doi:10.1145/3087801.3087815.

13 Weiming Feng and Yitong Yin. On local distributed sampling and counting. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 189–198, 2018. doi:10.1145/3212734.3212757.

14 Manuela Fischer and Mohsen Ghaffari. A simple parallel and distributed sampling technique: Local Glauber dynamics. In *32nd International Symposium on Distributed Computing (DISC)*, volume 121, pages 26:1–26:11, 2018. doi:10.4230/LIPICS.DISC.2018.26.

15 Xinyu Fu, Yitong Yin, and Chaodong Zheng. Locally-iterative $(\Delta+1)$-coloring in sublinear (in Δ) rounds. *arXiv preprint*, 2023. arXiv:2207.14458.

16 Marc Fuchs and Fabian Kuhn. Brief announcement: List defective colorings: Distributed algorithms and applications. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2023. doi:10.1145/3558481.3591319.

17 Joseph Gonzales, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel Gibbs sampling: From colored fields to thin junction trees. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 15, pages 324–332, 2011. URL: http://proceedings.mlr.press/v15/gonzalez11a/gonzalez11a.pdf.

18 G.E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002. doi:10.1162/089976602760128018.

19 G.E. Hinton, S. Osindero, and Y-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006. doi:10.1162/NECO.2006.18.7.1527.

20 G.E. Hinton and R.R. Salakhutdinov. Replicated softmax: an undirected topic model. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1607–1614, 2009. URL: https://proceedings.neurips.cc/paper/2009/hash/31839b036f63806cba3f47b93af8ccb5-Abstract.html.

21 Mark Jerrum. A very simple algorithm for estimating the number of k-colorings of a low-degree graph. *Random Structures & Algorithms*, 7(2):157–165, 1995. doi:10.1002/RSA.3240070205.

22 Michael I. Jordan, Jason D. Lee, and Yun Yang. Communication-efficient distributed statistical inference. *Journal of the American Statistical Association*, 2018. doi:10.1080/01621459.2018.1429274.

23 Christos Karras, Aristeidis Karras, Dimitrios Tsolis, Konstantinos C. Giotopoulos, and Spyros Sioutas. Distributed gibbs sampling and lda modelling for large scale big data management on pyspark. In *2022 7th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, pages 1–8, 2022. doi:10.1109/SEEDA-CECNSM57760.2022.9932990.

24 Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009. URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11886.

25 Fabian Kuhn. Weak graph colorings: Distributed algorithms and applications. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 138–144, 2009. doi:10.1145/1583991.1584032.

26 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

27 Hongyang Liu and Yitong Yin. Simple parallel algorithms for single-site dynamics. In *54th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1431–1444, 2022. doi:10.1145/3519935.3519999.

28 Kevin P. Murphy. *Machine learning: a probabilistic perspective*. MIT Press, 2012.

29 Moni Naor and Larry Stockmeyer. What can be computed locally? In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 184–193, 1993. doi:10.1145/167088.167149.

30 S. Osindero and G.E. Hinton. Modeling image patches with a directed hierarchy of Markov random fields. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1121–1128, 2008.

31 R. Salakhutdinov and G.E. Hinton. Deep Boltzmann machines. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 448–455, 2009.

32 R. Salakhutdinov, A. Mnih, and G.E. Hinton. Restricted Boltzmann machines for collaborative filtering. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, pages 791–798, 2007.

33 Jesús Salas and Alan D. Sokal. Absence of phase transition for antiferromagnetic Potts models via the Dobrushin uniqueness theorem. *Journal of Statistical Physics*, 86(3-4):551–579, 1997. doi:10.1007/BF02199113.

34 Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 257–266, 2010. doi:10.1145/1835698.1835760.

35 Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. In *Proceedings of the VLDB Endowment (PVLDB)*, 2010. doi:10.14778/1920841.1920931.

36 Alexander Terenin, Daniel Simpson, and David Draper. Asynchronous Gibbs sampling. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 144–154, 2020. URL: http://proceedings.mlr.press/v108/terenin20a.html.

37 Eric Vigoda. Improved bounds for sampling colorings. *Journal of Mathematical Physics*, 41(3):1555–1569, 2000. doi:10.1063/1.533196.

38 Feng Yan, Ningyi Xu, and Yuan Qi. Parallel inference for latent dirichlet allocation on graphics processing units. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 22, 2009. URL: https://proceedings.neurips.cc/paper/2009/hash/ed265bc903a5a097f61d3ec064d96d2e-Abstract.html.

39 Yuan Yang, Jianfei Chen, and Jun Zhu. Distributing the stochastic gradient sampler for large-scale LDA. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1975–1984, 2016. doi:10.1145/2939672.2939821.

40 Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. In *Proceedings of the VLDB Endowment (PVLDB)*, 2014. doi:10.14778/2732977.2733001.

Fever: Optimal Responsive View Synchronisation

Andrew Lewis-Pye
London School of Economics, UK

Ittai Abraham
VMWare Research, Herzliya, Israel

Abstract

View synchronisation is an important component of many modern Byzantine Fault Tolerant State Machine Replication (SMR) systems in the partial synchrony model. Roughly, the efficiency of view synchronisation is measured as the word complexity and latency required for moving from being synchronised in a view of one correct leader to being synchronised in the view of the next correct leader. The efficiency of view synchronisation has emerged as a major bottleneck in the efficiency of SMR systems as a whole. A key question remained open: Do there exist view synchronisation protocols with asymptotically optimal quadratic worst-case word complexity that also obtain linear complexity and responsiveness when moving between consecutive correct leaders?

We answer this question affirmatively with a new view synchronisation protocol for partial synchrony assuming partial initial clock synchronisation, called *Fever*. If n is the number of processors and t is the largest integer $< n/3$, then Fever has resilience t, and in all executions with at most $0 \leq f \leq t$ Byzantine parties and network delays of at most $\delta \leq \Delta$ after GST (where f and δ are unknown), Fever has worst-case word complexity $O(fn + n)$ and worst-case latency $O(\Delta f + \delta)$.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Systems, State Machine Replication

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.14

1 Introduction

Recent years have seen interest in developing protocols for State Machine Replication (SMR) that work efficiently at scale [10]. In concrete terms, this means looking to minimise the latency and the word complexity per consensus decision as a function of the number of participants n. Most commonly, this analysis takes place in the partial synchrony communication model, first suggested by Dwork, Lynch, and Stockmeyer [12]. The partial synchrony model forces the adversary to choose a point in time called the Global Stabilisation Time (GST) such that any message sent at time $\mathtt{t}$ must arrive by time $\max\{GST, \mathtt{t}\} + \Delta$. While Δ is known, the value of GST is unknown to the protocol. This model forms a practical compromise between the synchronous model (where all message delays are bounded by Δ), which is too optimistic, and the asynchronous model (where message delays are finite but unbounded), which is too pessimistic.

In a recent line of works [1, 14, 9] it has been shown that SMR can be solved with optimal resilience and with worst-case word complexity $O(n^2)$ after GST. Here, optimal resilience means being able to handle up to t Byzantine faults [12], where t is the greatest integer less than $n/3$. Given the lower bound of $\Omega(n^2)$ by Dolev and Reischuk [11], this bound on word complexity is tight.

The optimistic case. In practical settings, however, one typically cares not only about the worst-case, but also about the complexity and latency in the optimistic case when the actual (and unknown) number of failures f is less than the given bound t. Indeed, this is one of the principal motivations for considering the partial synchrony model. In the asynchronous model, where randomness is required after the initial cryptographic setup [13], one can

already achieve word complexity which is *expected* $O(n^2)$ per consensus decision [2]. In the partially synchrony model, the hope is that one may be able to define protocols which have worst-case complexity (providing cryptographic assumptions hold) which is $O(fn + n)$. Ideally, such protocols should also be *optimistically responsive*. Roughly, this means that the protocol should function at network speed if it turns out that $f = 0$: if $f = 0$, the protocol should be live during periods when message delay is less than the given bound Δ, but latency should be a function of the actual (unknown) message delay δ. This is important because the actual message delay δ may be much smaller than Δ when the latter value is conservatively set so as ensure liveness under a wide range of network conditions. More formally, we can say that a protocol is optimistically responsive if the latency after GST is $O(\Delta f + \delta)$ – a precise definition will be given in Section 2. An important feature of this definition is that latency is measured by the time until the first consensus decision *after* GST. Since GST is unknown, the definition applies to multi-shot protocols and cannot be satisfied by single-shot protocols. In particular, being optimistically responsive forces successive correct leaders after GST to complete successive consensus decisions at network speed.

Existing protocols for the partial synchrony model that give optimal resilience and worst-case complexity $O(n^2)$ do not satisfy the desired latency and complexity bounds described above. For such protocols [14, 9], the worst-case complexity is $O(n^2)$ but not $O(fn + n)$, while latency is $O(n\Delta)$.

The bottleneck is view synchronisation. Protocols for Byzantine Agreement and SMR typically divide the instructions into *views*, each with a dedicated leader that coordinates the protocol execution during that view. Since Hotstuff [19] shows how to achieve linear complexity within views, the remaining task is to define an efficient protocol that coordinates processors to execute instructions for the same view at the same time as each other. Accordingly, the task of defining efficient protocols for view synchronisation has become a principal focus [15, 16, 14, 9, 4], e.g. the protocols mentioned above, that achieve worst-case complexity $O(n^2)$ for Byzantine Agreement in the partial synchrony model, achieve this task by defining an appropriate method of view synchronisation.

Clock assumptions. Dwork, Lynch, and Stockmeyer [12] study several variants of partial synchrony. In the synchronous communication model, there is a known bound Δ on message delay. In the partially synchronous communication model, the known bound Δ on message delay only holds after the unknown time GST. Similarly, in the *synchronous processors* model, there is a known bound Φ on clock drift between correct processors. In the *partially synchronous processors* model, there is a known bound Φ on clock drift between correct processors after GST. Moreover, in the *completely synchronous processors model*, the clocks of all correct processors start at time 0 and there is no clock drift.

The setting we consider in this paper is the partially synchronous communication model with completely synchronous processors (as studied in Section 4 of [12]). In fact, our model allows for a slight relaxation and some clock drift (see below).[1] Our results do not hold if arbitrary clock drift can occur before GST, but, as described later, follow-up work already shows that our techniques can be used to improve state-of-the-art results under the assumption that arbitrary clock drift *can* occur prior to GST.

[1] As explained later, to obtain the results described here it suffices to assume the (potentially realistic) condition that there is some known bound on the different times at which correct processors begin the protocol execution and some known bound on clock drift for correct processors during periods of asynchrony.

From a practical perspective, we believe our result applies to a model that is realistic (at least in some scenarios of interest) and provides a compelling tradeoff: We show that, by using today's highly reliable hardware clocks, one can achieve view synchronisation with optimal communication complexity and latency, and for both the worst case and optimistic cases.

From a theoretical perspective, obtaining the positive results of this paper without strong clock synchronisation assumptions remains a challenging open question. We believe the fact that this has not been obtained to date, despite multiple publications and the centrality of this problem, may indicate that there is a natural barrier, and that in fact some type of clock synchronisation is required to obtain our results.

Our formal requirement on clock synchronisation, referred to as *partial initial clock synchronisation*, is defined in Section 2 and is, in fact, a relaxation of the partially synchronous communication model with completely synchronous processors. Using the assumption of partial initial clock synchronisation, we are able to define an innovative view synchronisation protocol in which the correct processors send at most $2n$ messages (combined) per view, and which is efficient in both the worst and optimistic cases.

The result. All terms in Theorem 1 will be formally defined in Section 2. Roughly, the worst-case word complexity of a view synchronisation protocol is the maximum number of words (each of maximum length determined by a security parameter) that need to be sent by correct processors during synchrony to synchronise all correct processors on a view with a correct leader. Similarly, the worst-case latency is the maximum time one has to wait during synchrony before all correct processors synchronise on a view with correct leader.

▶ **Theorem 1.** *Consider the partial synchrony model with maximum delay Δ after GST and with partial initial clock synchronisation. If t is the largest integer less than $n/3$, there exists a view synchronisation protocol with resilience t, such that for all executions with at most $0 \leq f \leq t$ Byzantine parties and network delays of at most $\delta \leq \Delta$ after GST (where f and δ are unknown):*

1. *The worst-case word complexity is $O(fn + n)$;*
2. *The worst-case latency is $O(\Delta f + \delta)$.*

In particular, for $f = 0$ this means $O(n)$ complexity and $O(\delta)$ latency and for $f = t$ this is $O(n^2)$ complexity and $O(\Delta n)$ latency.

Theorem 1 obtains worst-case quadratic communication, constant latency per malicious processor, and responsiveness between consecutive honest leaders. This resolves the main open question raised in Cogsworth[15].

Combined with Hotstuff, Theorem 1 gives an optimally resilient SMR protocol for the partial synchrony model that:

- **(i)** In the worst-case, requires $O(fn + n)$ words to be sent by correct processors after GST before confirmation of the first block of transactions after GST, and;
- **(ii)** Produces a first confirmed block of transactions after GST within time $O(\Delta f + \delta)$ of GST.

Since GST is unknown to the protocol, note that similar bounds then hold for the word complexity and latency between honestly produced confirmed blocks after GST.

Implications for results under standard clock assumptions. While Fever assumes "partial initial clock synchronisation" to achieve these optimal results, in fact, it has recently been shown[2] that Fever can also be combined with techniques introduced in [14], to produce a

[2] See `https://blog.chain.link/optimal-latency-and-communication-smr-view-synchronization/`

protocol named *Lumiere* that improves on the state-of-the-art *without* the requirement for partial initial clock synchronisation (but does not achieve the same results as Fever, described above). The results obtained by Lumiere and the trade-offs with Fever are discussed in Section 1.1.

1.1 Related work

Tendermint [7] showed how to use constant size messages for view-change. Casper FFG [8] extended this approach to allow pipelining. Hotstuff [19] extended these to define an SMR protocol achieving responsivenes and word complexity $O(n)$ within views, but did not rigorously establish an efficient technique for view synchronisation. In response to this, a number of papers have described view synchronisation protocols with different trade-offs.

Cogsworth [15] and Naor-Keidar [16] consider a setup in which leaders are chosen according to successive random permutations of the set of processors. They consider a static and *oblivious* adversary, who must choose GST without knowledge of the sequence of randomly chosen leaders, and which must also choose processors to corrupt at the start of the protocol execution without this knowledge. While we do not need to make use of any randomness (beyond that required for the initial cryptographic setup) to establish Theorem 1, we also consider such a setup for the purpose of apples-to-apples comparisons in Table 1 (in the "Expected Latency" and "Expected Complexity" columns). Both Cogsworth and Naor-Keidar achieve expected latency $O(\Delta)$ for such a static adversary, but this bound increases to $O(f^2\Delta + \delta)$ in the case that the adversary is adaptive, i.e. if the adversary can choose which processors to corrupt as the execution progresses (and with knowledge as to their choice of GST). The principal improvement of Naor-Keidar over Cogsworth is to decrease the expected complexity from $O(n^2)$ in the case of a static and oblivious adversary to $O(n)$. The expected complexity for Cogsworth becomes $O(fn^2 + n)$ in the case of an adaptive adversary, and the worst-case complexity is also $O(fn^2 + n)$. The expected complexity for Naor-Keidar becomes $O(f^2n + n)$ in the case of an adaptive adversary, and the worst-case complexity is also $O(f^2n + n)$. For a more detailed discussion of Cogsworth and Naor-Keidar, see the Appendix.

Table 1 View Synchroniser Comparisons.

Protocol	Expected Latency	Worst-case Latency	Expected Complexity	Worst-case Complexity	Partial Initial Clock Sync
Cogsworth	static adv: $O(\Delta)$ adaptive adv: $O(f^2\Delta + \delta)$	$O(f^2\Delta + \delta)$	static adv: $O(n^2)$ adaptive adv: $O(fn^2 + n)$	$O(fn^2 + n)$	Not needed
Naor-Keidar	static adv: $O(\Delta)$ adaptive adv: $O(f^2\Delta + \delta)$	$O(f^2\Delta + \delta)$	static adv: $O(n)$ adaptive adv: $O(f^2n + n)$	$O(f^2n + n)$	Not needed
Lewis-Pye	$O(n\Delta)$	$O(n\Delta)$	$O(n^2)$	$O(n^2)$	Not needed
Raresync	$O(n\Delta)$	$O(n\Delta)$	$O(n^2)$	$O(n^2)$	Not needed
Fever (this paper)	static adv: $O(\Delta)$ adaptive adv: $O(f\Delta + \delta)$	$O(f\Delta + \delta)$	static adv: $O(n)$ adaptive adv: $O(fn + n)$	$O(fn + n)$	Needed

In Table 1, we assume the *bound* t on the number of Byzantine processors is the largest integer less than $n/3$, so that $t = \Theta(n)$, while $0 \leq f \leq t$ is the *actual* number of Byzantine processors. "Complexity" means "word complexity". Both latency and word complexity are defined in Section 2, as is the "partial clock synchronisation" condition. We only distinguish explicitly between a static and adaptive adversary when this changes the corresponding bound.

While the published version of Hotstuff [19] did not describe any efficient method for view synchronisation, the original version (posted on the arXiv [1]) did roughly outline an approach to meeting the $O(n^2)$ worst-case complexity bound of Dolev-Reischuk. This approach was made precise and rigorously proved in [14] and [9]. These papers described view synchronisation protocols which we will refer to as "Lewis-Pye" and "Raresync" respectively. A disadvantage of these protocols over Fever (which we describe in this paper) is that they both have worst-case latency $O(n\Delta)$, as opposed to $O(f\Delta + \delta)$ for Fever, and worst-case complexity $O(n^2)$, as opposed to $O(fn + n)$ for Fever.

Thus far, we have focused on view synchronisation protocols for the partial synchrony model. It should be emphasized that the stronger efficiency bounds for Fever are achieved via a novel view synchronisation protocol combined with stricter assumptions on initial clock synchronisation (as made precise in Section 2).

It is well-known that protocols in the asynchronous model can achieve *expected* complexity $O(n^2)$ per consensus decision (e.g. see [2]). The trade-off when compared with the protocol we present here (when combined with Hotstuff) is that such asynchronous protocols do not require synchronous intervals to be live, but still have complexity $O(n^2)$ in the case that $f = 0$.

In [18], Spiegelman describes a (single-shot) protocol for Byzantine Agreement which is designed to operate efficiently under both synchronous and asynchronous conditions. The protocol achieves expected $O(n^2)$ complexity in asynchrony and $O(fn + n)$ in synchrony. In the partial synchrony model, the expected complexity after GST remains $O(n^2)$ in the case that $f = 0$ (as opposed to $O(n)$ for Fever). If one isolates the part of the (single-shot) protocol which is designed to function under synchrony, and applies just this protocol in the partial synchrony model, then there is no need for view synchronisation, but the protocol is not then optimistically responsive (e.g. successive honest leaders do not produce successive consensus decisions in time $O(\delta)$ during synchrony).

A recent sequence of papers by Bravo, Chockler, and Gotsman [4, 5, 6] describe a modular framework for the analysis of view-based SMR protocols. The aim of those papers is complementary to and different than our aim here. While those authors describe a general framework and are less concerned with establishing optimal results in terms of complexity and latency (such as those described here), our aim is to describe a specific view synchronisation protocol achieving state-of-the-art efficiency. An advantage of the approach described by those authors is that it allows for a PBFT-style approach to view change, whereby a single leader may persist until correct processors request a change in leader. By contrast, the view synchronisation protocol we describe here has processors automatically pass through views with different leaders. So, this is at least one sense in which the approach described by Bravo et. al. is more general than what we describe here.

Lumiere. As noted in the introduction, it has recently been shown that Fever can be combined with techniques from [14] to give a protocol named Lumiere, which significantly improves on the state-of-the-art without the need for partial initial clock synchronisation. Roughly, Lumiere divides the views into sets of views called *epochs* and uses Fever for synchronization within epochs, while requiring a heavier synchronization procedure for movement between epochs. While the Lewis-Pye [14] protocol achieves $O(n^2)$ worst-case complexity, a significant drawback is that, even after the first synchronisation point (i.e. even after the first time after GST when all correct processors are synchronised on a view with correct leader), a single faulty leader can subsequently cause a delay $O(n\Delta)$ between consensus decisions. Lumiere overcomes this issue – while the latency and complexity figures for Lumiere with respect to the measures used in Table 1 (which concern the latency

and complexity until the first synchronisation point) are the same as for the Lewis-Pye protocol, Lumiere is optimistically responsive and gives optimal performance after the first synchronisation point, without the need for partial initial clock synchronisation. If $\mathtt{t}$ is any time after the first synchronisation point, and if latency is given by the time between $\mathtt{t}$ and the first consensus decision after $\mathtt{t}$, then Lumiere has latency $O(f\Delta + \delta)$ (while maintaining optimal worst-case complexity $O(n^2)$).

2 The setup

We consider a set $\Pi = \{p_0, \dots, p_{n-1}\}$ of n processors, and let t be the largest integer less than $n/3$. Each processor p_i is told i as part of its input. For the proof of Theorem 1, we assume an adaptive adversary that is able to choose at most t processors to corrupt as the execution progresses. A processor that is corrupted by the adversary at any point in the execution is referred to as *Byzantine*, and may behave arbitrarily once corrupted. Processors that are not Byzantine are *correct*. We let f denote the actual number of Byzantine processors.

Cryptographic assumptions. Our cryptographic assumptions are standard for papers on this topic. Processors communicate by point-to-point authenticated channels. We use a cryptographic signature scheme, a public key infrastructure (PKI) to validate signatures, and a threshold signature scheme [3, 17]. The threshold signature scheme is used to create a compact signature of m-of-n processors, as in other consensus and view synchronisation protocols [19]. In this paper, either $m = t+1$ or $m = n-t$. The size of a threshold signature is $O(\kappa)$, where κ is a security parameter, and does not depend on m or n. We assume a computationally bounded adversary. Following a common standard in distributed computing and for simplicity of presentation (to avoid the analysis of negligible error probabilities), we assume these cryptographic schemes are perfect, i.e. we restrict attention to executions in which the adversary is unable to break these cryptographic schemes.

Communication. As noted above, processors communicate using point-to-point authenticated channels. We consider the standard partial synchrony model, whereby a message sent at time $\mathtt{t}$ must arrive by time $\max\{GST, \mathtt{t}\} + \Delta$. While Δ is known, the value of GST is unknown to the protocol. The adversary chooses GST and also message delivery times, subject to the constraints already defined.

According to the definition above, messages sent prior to GST may be significantly delayed, but are not lost. We only use the assumption that messages are not lost, however, when analysing the word complexity of reaching the *first* synchronisation after GST. Our view synchronisation protocol works without this assumption, and the assumption could be dropped if one was to consider complexity measures which are less strict than that we consider here, such as that in [15].

Partial Initial Clock Synchronisation. To specify our assumptions on the times at which honest processors begin the protocol execution, let $c(p)$ denote the value of processor p's clock. At any point $\mathtt{t}$ in an execution, let $T(\mathtt{t}) := \{c(p) : \ p \text{ is correct}\}$. In particular, this means that $T(0)$ is the set of all clock values for correct processors at the start of the protocol execution. Our required condition regarding initial clock synchronisation is that, for some known bound Γ:

($\dagger_{\Gamma,\mathtt{0}}$) For any $\mathtt{c} \in T(0)$:

$$|\{\mathtt{c}' \in T(0) : \ \mathtt{c}' \geq \mathtt{c} - \Gamma\}| \geq t + 1.$$

Recall that t is the bound on the number of Byzantine processors. So, the condition above says that, for each correct processor p, there are at least t other correct processors whose clocks (may be arbitrarily ahead of p's clock but) are at most Γ behind p's clock. Specifically, this is really a condition on the most advanced clock of an correct processor. If p's clock is the most advanced amongst correct processors, then we require that there are at least t correct processors whose clocks are at most Γ behind p's clock. Another way of looking at this is that all correct processors begin the protocol execution with their local clock set to 0, and that if p is the first correct processor to begin the protocol execution (while other clocks may still be negative, so that those processors are still waiting to start), then at least t other correct processors begin the protocol execution within time Γ. Note that this condition does not place any bound on the maximum difference between the clocks of correct processors.

For the sake of simplicity, we will also initially assume that all correct processors have identical clock speeds. Then, in Section 5, we will consider realistic relaxations of this condition that suffice to give our results.

The underlying protocol. We suppose view synchronisation is required for some underlying protocol (such as Hotstuff) with the following properties:

- **Views**. Instructions are divided into views. Each view v has a designated *leader*, denoted $\texttt{lead}(v)$. For some parameter $k \geq 3$ (which can be chosen to suit the protocol designer's needs), we suppose views are grouped into sets of k, so that the leader[3] for view v is processor p_i where $i := \lfloor v/k \rfloor \bmod n$. If $v \bmod k = 0$, then v is called "initial".
- **Quorum certificates**. The successful completion of a view v is marked by all processors receiving a *Quorum Certificate* (QC) for view v. The QC is a threshold signature of length $O(\kappa)$ (for the security parameter κ that determines the length of signatures and hash values) combining $n - t$ signatures from different processors testifying that they have completed the instructions for the view. In a chained implementation of Hotstuff, for example, the leader will propose a block, processors will send votes for the block to the leader, who will then combine those votes into a QC and send this to all processors. Alternatively, one could consider a (non-chained) implementation of Hotstuff, in which the relevant QC corresponds to a successful third round of voting. Note that the production of QCs is not a restrictive assumption, since if it is not satisfied one can easily amend the instructions of the protocol so that it is.
- **Sufficient time for view completion**. We suppose there exists some known $x \geq 2$ such that if $\texttt{lead}(v)$ is correct, if (the global time) $\texttt{t} \geq$ GST, and if at least $n - t$ correct processors are in view v from time $\texttt{t}$ until either they receive a QC for view v or until $\texttt{t} + x\delta$, then all correct processors will receive a QC for view v by time $\texttt{t} + x\delta$, so long as all messages sent by correct processors while in view v are received within time $\delta \leq \Delta$. For the sake of simplicity, we assume Γ from the definition of "partial clock synchronisation" is equal to $x\Delta$– if these values differ then one can just take the maximum of the two values.

The view synchronisation task. For Γ as above, we must ensure:

1. If a correct processor is in view v at time $\texttt{t}$ and in view v' at $\texttt{t}' \geq \texttt{t}$, then $v' \geq v$.
2. There exists some correct $\texttt{lead}(v)$ and $\texttt{t} \geq GST$ such that each correct processor is in view v from time $\texttt{t}$ until either it receives a QC for view v or until $\texttt{t} + \Gamma$.

[3] These assumptions are made for the purpose of proving Theorem 1. In verifying the bounds given in Table 1, we will also consider the possibility of random leader selection.

Condition (1) above is required by standard view-based SMR protocols to ensure consistency. Since GST is unknown to the protocol, condition (2) suffices to ensure the successful completion of infinitely many views with correct leaders. By a *view synchronisation protocol*, we mean a protocol which determines when processors enters views and which satisfies conditions (1) and (2) above.

Complexity measures. Our proofs are quite robust to the precise notions of latency and word complexity considered, and will hold for any of the definitions used in previous papers on the topic such as [15, 16, 4]. For the sake of concreteness, we fix complexity measures which are as strict as possible, and note that if we were to adopt the more relaxed measures used in [15], for example, then we could weaken the requirement that messages sent before GST are not lost.

By a "word", we mean a message of length $O(\kappa)$, where κ is the security parameter determining the length of signatures and hash values. We make the following definitions. Let $\mathtt{t}^*$ be the least time $> GST$ at which the underlying protocol has some correct $\mathtt{lead}(v)$ produce a QC for view v (if there exists no such time, set $\mathtt{t}^* := \infty$). The worst-case word complexity is the maximum number of words sent by correct processors (combined) between time $GST + \Delta$ and $\mathtt{t}^*$. The worst-case latency is the maximum possible value of $\mathtt{t}^* - GST$.

Defining optimistic responsiveness. We do not need to define optimistic responsiveness to establish Theorem 1. For the sake of concreteness, however, we can define our view synchronisation protocol to be optimistically responsive if the worst-case latency is $O(f\Delta+\delta)$, where f is the (unknown) number of Byzantine processors and $\delta \leq \Delta$ is the actual (unknown) bound on message delay after GST. Note that our latency and complexity measures above concern the time and word complexity until the first consensus decision produced by a correct leader strictly *after* GST, and imply that single-shot protocols cannot be optimistically responsive.

3 The protocol

Recall that views are grouped into sets of k, so that the leader for view v is processor $\lfloor v/k \rfloor \bmod n$. If $v \bmod k = 0$, then v is called "initial". To synchronise processors, we have a predetermined "clock-time" corresponding to each view: The clock-time corresponding to view v is $\mathtt{c}_v := \Gamma v$.

The rough idea is that, at certain points in the execution (and to satisfy optimistic responsiveness), we have processors instantaneously forward their clock to some clock-time $\mathtt{c}_v$ and enter view v. We do this in such a way to ensure that, if p is the correct processor whose local clock is most advanced, then there are always at least t other correct processors whose local clocks are at most Γ behind p's clock. This will suffice to ensure correct leaders are able to synchronise all correct processors after GST.

The instructions are defined simply as follows:

When processors enter views. Recall that, at any point in the execution, $c(p)$ is the value of processor p's clock. If v is initial, then p enters view v when $c(p) = \mathtt{c}_v$. If v is not initial, then p enters view v if it is presently in a view $< v$ and it receives a QC (formed by the underlying protocol) for view $v - 1$.

View Certificates. When a correct processor p enters a view v which is initial, it sends a `view` v message to `lead`(v). This message is just the value v signed by p. Once `lead`(v) receives $t+1$ `view` v messages from distinct processors, it combines these into a single threshold signature, which is a view certificate (VC) for view v, and sends this VC to all processors.[4]

When processors forward clocks. At any point in the execution, if a correct processor p receives a QC for view $v-1$ (formed by the underlying protocol) or a VC for view v, and if $c(p) < \mathtt{c}_v$, then p instantaneously forwards their clock to $\mathtt{c}_v$.

Pseudocode for the protocol is given in Algorithm 1.

Algorithm 1 The instructions for processor p.

Local variables
$c(p)$, intially 0 ▷ This is the value of p's clock
v, initially 0 ▷ This is the present view of p.

Global parameters
n ▷ Number of processors
t ▷ Largest integer $< n/3$
$k := 3$ ▷ Can take larger values.
$\mathtt{c}_{v'} := v'\Gamma$, $v' \in \mathbb{N}_{\geq 0}$ ▷ Defines clock times
`lead`$(v') := p_i$ for $i = \lfloor v'/k \rfloor \bmod n$ and $v' \in \mathbb{N}_{\geq 0}$ ▷ Specifies leaders

Upon $c(p) == \mathtt{c}_{v'}$ for v' initial
    Set $v := v'$
    Send a `view` v message to `lead`(v)

Upon first seeing a QC for view $v' \geq v$
    Set $v := v' + 1$
    If $c(p) < \mathtt{c}_{v'+1}$ set $c(p) := \mathtt{c}_{v'+1}$

Upon first seeing a VC for initial view $v' > v$
    Set $v := v'$
    If $c(p) < \mathtt{c}_{v'}$ set $c(p) := \mathtt{c}_{v'}$

If $p ==$ `lead`(v') for $v' \geq v$ **then**
    Upon first seeing `view` v' messages from $t+1$ distinct processors
        Form a VC for view v' and send to all processors

The informal intuition behind the protocol. Partial initial clock synchronisation requires that, at the start of the protocol execution, and if p is the correct processor whose local clock is most advanced, there are at least t other correct processors whose local clocks are at most Γ behind p's clock. The protocol above is specified to ensure this condition remains

[4] It is convenient throughout to assume that when a leader sends a message to all processors, this includes itself.

true throughout the execution – see Section 4 for a simple proof. This will be easily seen by checking that the condition can never be violated by the forwarding of clocks. Now suppose that $\texttt{lead}(v)$ is correct and that a correct processor, p say, is the first to enter view v after GST at time $\texttt{t}$. Our condition on local clocks, described above, means that t other correct processors will also enter view v within a short time. Since $\texttt{lead}(v)$ only requires $t+1$ signatures to form a VC for view v, all correct processors will then receive a VC for view v within a short time. The underlying protocol will then have $\texttt{lead}(v)$ put together a QC for view v.

4 The proofs

It is immediate from the instructions that if a correct processor enters a view v then it cannot subsequently enter any lower view.

Recall that, at any point $\texttt{t}$ in an execution, $T(\texttt{t}) := \{c(p) : \ p \text{ is correct}\}$. Our condition for "partial initial clock synchronisation" required that a certain condition $(\dagger_{\Gamma,0})$ holds at the start of the protocol execution. This condition requires that if p is the correct processor whose local clock is most advanced, then at least t other correct processors have clocks that are at most Γ behind p's clock. The key to the proof is to show that an analogous condition then holds at all times.

▶ **Lemma 1.** *For all* $\texttt{t}$ *the following condition holds:*
$(\dagger_{\Gamma,\texttt{t}})$ *For any* $\texttt{c} \in T(\texttt{t})$:

$$|\{\texttt{c}' \in T(\texttt{t}) : \ \texttt{c}' \geq \texttt{c} - \Gamma\}| \geq t+1.$$

Before proving Lemma 1, we note that the lemma does *not* place any bound on the maximum difference between the local clocks of correct processors. In fact, even if all clocks are initially perfectly synchronised, the local clocks of two correct processors can move arbitrarily far apart prior to GST. Nevertheless, the fact that $(\dagger_{\Gamma,\texttt{t}})$ holds for all $\texttt{t}$ will suffice to establish Theorem 1.

Proof (Lemma 1). Since the local clocks of correct processors only ever move forward, it follows that at any point in an execution, if a correct processor p has already contributed to a QC or a VC for view v, then $c(p) \geq \texttt{c}_v$. To prove that $(\dagger_{\Gamma,\texttt{t}})$ holds for all $\texttt{t}$, suppose towards a contradiction that there is a first point of the execution, $\texttt{t}$ say, for which there exists some correct processor p such that $|\{\texttt{c} \in T(\texttt{t}) : \ \texttt{c} \geq c(p) - \Gamma\}| < t+1$. Then p must forward its clock at $\texttt{t}$. There are two possibilities:

1. p forwards its clock because it receives a VC for some view v with $\texttt{c}_v > c(p)$. In this case, there must exist at least one correct processor $p' \neq p$ which contributed to the VC for view v. By the choice of $\texttt{t}$, when p' contributed to the VC at $\texttt{t}' \leq \texttt{t}$ we had $|\{\texttt{c} \in T(\texttt{t}') : \texttt{c} \geq c(p') - \Gamma\}| \geq t+1$. Since $c(p') \geq \texttt{c}_v$ when it contributed to the VC, and since $c(p) = \texttt{c}_v$ at $\texttt{t}$, at $\texttt{t}$ we have that $|\{\texttt{c} \in T(\texttt{t}) : \texttt{c} \geq c(p) - \Gamma\}| \geq t+1$ also, which gives the required contradiction.
2. p forwards its clock because it sees a QC. In this case, at least $t+1$ correct processors must have contributed to the QC, meaning that their clocks are at most Γ behind p's clock, which directly gives the required contradiction. ◀

▶ **Lemma 2.** *If* v *is initial and* $\texttt{t}$ *is the first time any correct processor enters a view* $\geq v$*:*

(i) *A correct processor enters view* v *at* $\texttt{t}$*;*
(ii) *No correct processor enters any view* $v' > v$ *at* $\texttt{t}$*, and;*
(iii) $c(p) \leq \texttt{c}_v$ *for all correct* p *at* $\texttt{t}$*.*

Proof. Consider the first time any correct processor p enters a view $v' \geq v$. It cannot be because p sees a VC for view v', because some correct processor must then have contributed to that VC and already have been in view v'. It cannot be because p sees a QC for view $v' - 1 > v - 1$, because $t + 1$ correct processors must have already contributed to that QC (noting that p is the first to enter any view $\geq v$). It follows that the first view $v' \geq v$ entered by any correct processor is v. When the first correct processor p enters view v we have $c(p) = \mathtt{c}_v$ (either simply because it reaches this value, or else because p sees a QC for view $v - 1$), and that $c(p') \leq \mathtt{c}_v$ for all correct p' at this point. ◀

▶ **Definition 3.** *Let $\mathtt{t}(v)$ be the first time at which a correct processor enters view v.*

Since correct processors enter an unbounded number of views, it follows from Lemma 2 that if v is initial then $\mathtt{t}(v)\downarrow$ and[5] $\mathtt{t}(v') > \mathtt{t}(v)$ whenever $v' > v$ and $\mathtt{t}(v')\downarrow$.

Note also that if v is initial then, for $j \in (0, k)$, a QC for view $v + j$ cannot be formed prior to the formation of a QC for view $v + j - 1$. This follows because (since v is initial, and for j in the given range) correct processors do not enter view $v + j$ without seeing a QC for view $v + j - 1$. The next lemma will be used to show that correct processors spend a sufficiently long time in each view that a correct leader after GST will be able to produce QCs.

▶ **Lemma 4.** *Suppose v is initial. For each $j \in [0, k)$, let $\mathtt{s}_j$ be the first time (if there exists such) at which a correct processor sees a QC for view $v+j$. The first time at which any correct processor enters view $v+k$ is the minimum amongst the values $\{\mathtt{t}(v)+k\Gamma\} \cup \{\mathtt{s}_j + (k-1-j)\Gamma : \mathtt{s}_j \downarrow\}$.*

Proof. By Lemma 2, some correct processor p enters v at $\mathtt{t}(v)$, and all correct processors p' have $c(p') \leq \mathtt{c}_v$ at this point. As we reasoned in the proof of Lemma 2, it cannot be the case that the first time any correct processor enters a view $v' \geq v + k$ it is because it sees a VC for view v' or a QC for $v' - 1 \geq v + k$. It follows that the first time a correct processor p enters view $v + k$ it is because its local clock has reached $\mathtt{c}_{v+k}$. This happens either because p saw a QC for view $v + j$ ($j \in [0, k)$) and then time $(k - 1 - j)\Gamma$ passed (meaning zero time if $j = k - 1$), or else because p was the first correct processor to enter view v and time $k\Gamma$ passed since that point. ◀

With Lemmas 1, 2 and 4 in place, the basic intuition behind the idea that a correct leader will produce a QC after GST is clear. Let $\mathtt{lead}(v)$ be correct and such that no correct processor enters view v prior to GST. From Lemma 2, it follows that no correct processor enters any view $v' > v$ prior to $\mathtt{t}(v)$. By Lemma 1, at least $t + 1$ correct processors will have entered view v within time Γ – by Lemma 4, no correct processor will be in any view $> v$ prior to the first of $\mathtt{t}(v) + k\Gamma$ or else the formation of a QC for view v. The correct processor $\mathtt{lead}(v)$ will then form a VC for view v, and all correct processors will be in view v by time $\mathtt{t} + \Gamma + 2\Delta$ unless a QC for view v has already been formed by this point. This means that all processors will receive a QC for view v by time $\mathtt{t} + 2\Gamma + 2\Delta$. Since $\Gamma \geq 2\Delta$ and $k \geq 3$, this suffices.

Now let us see the details. In the below, we prove more than the fact that a correct $\mathtt{lead}(v)$ will produce a QC for one of the views in $[v, v + k)$. We show that $\mathtt{lead}(v)$ will produce QCs for multiple successive views if k is large enough, since this will be useful in some implementations (such as chained implementations of Hotstuff etc).

[5] We write $x\downarrow$ to denote that the variable x is defined.

▶ **Lemma 5.** *Suppose v is initial, $\texttt{lead}(v)$ is correct, and that $\texttt{t}(v) \geq GST$. Then correct processors will see QCs for all views in $[v, v+k-2)$ before entering view $v+k$.*

Proof. By Lemma 2, no correct processor has entered any view $v' > v$ at $\texttt{t}(v)$. By Lemma 1, $(\dagger_{\Gamma,\texttt{t}(v)})$ is satisfied, which means at least $t+1$ `view` v messages will have been sent to $\texttt{lead}(v)$ by $\texttt{t}(v)+\Gamma$ – by Lemma 4, no correct processor will be in any view $> v$ prior to the point at which these $t+1$ `view` v messages have been sent to $\texttt{lead}(v)$. Then $\texttt{lead}(v)$ will have sent out a VC for view v by $\texttt{t}(v)+\Gamma+\Delta$, which will be received by all correct processors by time $\texttt{t}(v)+\Gamma+2\Delta$. It then follows from Lemma 4, and since $\Gamma \geq 2\Delta$, that a QC for each view $j \in [0, v+k-2)$ will be seen by all correct processors by time $\texttt{t}(v)+\Gamma+2\Delta+(j+1)\Gamma$, prior to any point at which a correct processor enters view $v+k$. ◀

▶ **Lemma 6.** *The worst-case word complexity is $O(fn+n)$ and the worst-case latency is $O(\Delta f+\delta)$.*

Proof. We deal with the word complexity first. Let p be the correct processor whose clock is most advanced at GST (breaking ties arbitrarily). Suppose p is in view v at GST. Let v_0 be the greatest initial view $< v$ such that $\texttt{lead}(v_0) \neq \texttt{lead}(v)$ and $\texttt{lead}(v_0)$ is correct. Let v_1 be the least initial view $> v$ such that $\texttt{lead}(v_1)$ is correct. Since no correct processor will enter any view $> v_0$ prior to the least of $\texttt{t}(v_0)+k\Gamma$ or the first time at which a correct processor sees a QC for view v, and since $(\dagger_{\Gamma,\texttt{t}(v_0)})$ is satisfied, $\texttt{lead}(v_0)$ must have sent a VC for view v_0 to all processors prior to GST. All correct processors will therefore be in at least view v_0 by $GST+\Delta$. Lemma 5 shows that all correct processors will see a QC for view v_1 before entering view v_1+k. Let f^* be the number of Byzantine leaders for initial views in the interval (v_0, v_1). Correct processors will send a maximum of $2(f^*+3)n$ many `view` messages (combined) between $GST+\Delta$ and the time at which $\texttt{lead}(v_1)$ produces a QC for view v_1. If the underlying protocol has correct processors send $O(n)$ messages per view (e.g. Hotstuff), then the underlying protocol will also have correct processors send $O((f^*+3)n)$ messages during this interval. So the worst-case word complexity is $O(fn+n)$, as required.

Next, we consider the worst-case latency. If $f > 0$, then it suffices to observe that $\texttt{lead}(v_1)$ will produce a QC for view v_1 by time $GST+k(f^*+3)\Gamma$. So suppose $f = 0$ and consider the number d of correct processors in view v at GST. If $d \geq t+1$, then $\texttt{lead}(v)$ will produce a QC for view v within time $O(\delta)$ according to our assumptions on the underlying protocol, unless at least $t+1$ processors enter view $v+k$ before this occurs. In the latter case, $\texttt{lead}(v+k)$ will produce a QC for view $v+k$ within time $O(\delta)$. If $d < t+1$, then (the previous leader) $\texttt{lead}(v_0)$ will produce a QC for view v within time $O(\delta)$, unless at least $t+1$ processors enter view v before this occurs. In the latter case, $\texttt{lead}(v)$ will produce a QC for view v within time $O(\delta)$. ◀

Lemma 6 completes the proof of Theorem 1. We finish this section by justifying the entries of Table 1, which state that the expected latency is $O(\Delta)$ and the expected word complexity is $O(n)$ with a static adversary according to the model of [15]. According to this model, leaders are given by successive random permutations of the set of all processors. The adversary is static and *oblivious*, which means that they must choose which processors to corrupt at the start of the protocol execution without knowledge as to the random sequence of leaders, and must also choose GST without this knowledge. In this case, the expected value f^* from the proof of Lemma 6 is $O(1)$, which means we get expected latency $O(\Delta)$ and expected word complexity $O(n)$, as required.

5 Tying up loose ends

5.1 A note on optimistic responsiveness and Byzantine leaders

In the proof of Lemma 6, it was only actually the number of Byzantine *leaders* before the first correct leader after GST that mattered (rather than the total number of Byzantine parties) in establishing that the worst-case word complexity is $O(fn+n)$. The proof that the worst-case latency is $O(f\Delta+\delta)$ was somewhat more subtle, and considered the total number of Byzantine processors. Roughly, the difficulty occurs when none of the relevant leaders are Byzantine, but a correct processor has just entered initial view v at GST, while all other correct processors are still in previous views. In this scenario, the previous leader cannot produce a QC (if t parties are Byzantine while not in their role as leader). Meanwhile, the leader for view v has to wait time $\Gamma+\delta$ before producing a VC. As we argued in the proof of Lemma 6, this is not an issue if we let f count the total number of Byzantine parties.

Once a first correct leader produces a QC after GST, however, this subtlety is no longer relevant. Let $\mathtt{lead}(v)$ be correct and such that $\mathtt{t}(v) \geq GST$. Let v' be the least view with correct leader $> v$ and suppose that the number of initial views with Byzantine leader in the interval (v, v') is f^*. Then the proof of Lemma 6 is easily modified to show that correct processors send $O(f^*n+n)$ many words between the times at which $\mathtt{lead}(v)$ and $\mathtt{lead}(v')$ produce QCs, and that the time between these events is $O(f^*\Delta+\delta)$.

5.2 Revisiting the assumptions regarding clock synchronisation

In Section 2 we assumed that all processors have identical clock speeds. We now consider to what extent we can relax this condition. As we do so, we also consider how realistic are the required assumptions in the context of reasonable bounds on network delays, the length of periods of asynchrony etc., and in a context where atomic clocks are available for use by processors. Recall that atomic clocks can reasonably be assumed to have error less than 1 second every 100 million years.[6]

In the partial synchrony model it is only for the sake of technical convenience that we consider a single period of asynchrony and then a single period of synchrony after GST. In reality, we are interested in contexts where network conditions oscillate between synchrony and asynchrony. We require our protocols to maintain consistency during periods of asynchrony, and to be live during periods of synchrony. To ensure that our analysis extends to such a scenario, let us therefore consider our requirements as the network oscillates between periods of sychrony and asynchrony in this fashion.

Fix $k := 3$. A similar analysis will also apply for larger values of k. Let us say an open interval $(\mathtt{t}, \mathtt{t}')$ is synchronous if every message sent in this interval arrives within time Δ. Let $\ell := 3(t+3)\Gamma$, where t is the bound on the number of Byzantine processors. If $(\dagger_{\Gamma,\mathtt{t}'})$ holds for all $\mathtt{t}' \in I := (\mathtt{t}, \mathtt{t}+\ell)$ and if I is synchronous, the proofs of Section 4 established that some correct leader will produce a QC during interval I and send this to all processors. With this in mind, we inductively define a sequence of times $(\mathtt{t}_i)_{i\geq 0}$ as follows:

- Let $\mathtt{t}_0$ be the least that $(\mathtt{t}_0, \mathtt{t}_0+\ell)$ is synchronous.
- Given $\mathtt{t}_i$, let $\mathtt{t}_{i+1}$ be the least $\mathtt{t} \geq \mathtt{t}_i + \ell$ such that $(\mathtt{t}, \mathtt{t}+\ell)$ is synchronous.

[6] See, for example, `https://en.wikipedia.org/wiki/Atomic_clock`

We suppose that every $\mathtt{t}_i$ is defined. For concreteness, it is also useful to stipulate some specific values – a similar argument will hold for comparable values:

- Suppose $\Delta = 1$ second.
- Suppose $\mathtt{t}_0 < 10^5$ years and the maximum value $\mathtt{t}_{i+1} - \mathtt{t}_i$ is less than 10^5 years.
- For view v, suppose $\mathtt{t}$ is the first time at which $t+1$ correct processors are in view v, and that $\mathtt{t}'$ is the first time at which a correct processor sees a QC for view v. Define $\mathtt{u}_v := \mathtt{t}' - \mathtt{t}$. When $\mathtt{u}_v$ is defined, we suppose it always has at least the mininum value $\mathtt{u}$. For the sake of concreteness, we suppose $\mathtt{u} = 10^{-2}$ seconds.
- Suppose that $(\dagger_{\Delta,0})$ holds, and that $\Gamma = 2\Delta$.

We note that the assumptions above are weak: In particular, we assume only that a synchronous interval exists every 10^5 years. Then we claim $(\dagger_{\Gamma,\mathtt{t}})$ holds for all $\mathtt{t}$ – this is the condition required to ensure that every interval $(\mathtt{t}_i, \mathtt{t}_i + \ell)$ has a correct leader produce a QC. Towards a contradiction, suppose there exists a least value i^* such that $(\dagger_{\Gamma,\mathtt{t}})$ fails to hold for some $\mathtt{t}^*$ in the interval $[\mathtt{t}_{i^*}, \mathtt{t}_{i^*+1})$. For each $\mathtt{t}$, let $\Gamma(\mathtt{t})$ be the smallest Γ' such that $(\dagger_{\Gamma',\mathtt{t}})$ holds. Note that:

- Every interval $(\mathtt{t}_i, \mathtt{t}_i + \ell)$ such that $i < i^*$ has at least one correct processor synchronise the clocks of correct processors to within time Δ, i.e. $(\dagger_{\Delta,\mathtt{t}})$ holds for some $\mathtt{t}$ in this interval.
- When a correct processor sees a QC and forwards its clock at $\mathtt{t}$, this may cause $\Gamma(\mathtt{t})$ to increase, e.g. if the leader is Byzantine and only sends the QC to certain processors, or if the QC is not sent during a synchronous interval. In this case, however, the maximum value of $\Gamma(\mathtt{t})$ is still at most $\Gamma - \mathtt{u}$.
- If a processor forwards its clock because it sees a VC at $\mathtt{t}$, this does not increase $\Gamma(\mathtt{t})$.

Define $\mathtt{t} := \mathtt{t}_{i^*-1} + \ell$ if $i^* \neq 0$, and define $\mathtt{t} := 0$ if $i^* = 0$. Let $\mathtt{t}^*$ be defined as above. We conclude that $\Gamma(\mathtt{t})$ is at most $\max\{\Delta, \Gamma - \mathtt{u}\}$, to within a small error term which is the maximum drift of clocks within an interval of length ℓ. Since we suppose $\mathtt{u} = 10^{-2}$ seconds, since our clocks have drift at most 1 second every 100 million years, and since some clocks may drift slow while others drift fast, this means that $\mathtt{t}^* - \mathtt{t} > 5 \times 10^5$ years. This gives the required contradiction, since we assumed above that $\mathtt{t}_0 < 10^5$ years and $\mathtt{t}_{i+1} - \mathtt{t}_i$ is less than 10^5 years for all i.

6 Concluding comments

We have defined Fever, which is a novel view synchronisation protocol. If n is the number of processors and t is the largest integer $< n/3$, then Fever has resilience t, and in all executions with at most $0 \leq f \leq t$ Byzantine parties and network delays of at most $\delta \leq \Delta$ after GST (where f and δ are unknown), Fever has worst-case word complexity $O(fn + n)$ and worst-case latency $O(\Delta f + \delta)$. This improves significantly on the state-of-the-art.

The trade-off is that Fever requires greater assumptions than previous view synchronisation protocols regarding the drift of clocks prior to GST. We have argued in Section 5 that there are scenarios in which our required assumptions are reasonable. Atomic clocks can now be purchased for a few thousand US dollars, and we showed that under reasonable assumptions regarding network latency etc., a system implementing Fever will be able to handle periods of asynchrony of the order of 10^5 years. Of course, this is more than is reasonably required, and so even the use of less accurate clocks may suffice in many scenarios.

We also noted, in Section 1.1, that it has been shown that Fever can be combined with other techniques to produce a protocol called Lumiere that improve on the state-of-the-art without the need for partial initial clock synchronisation. Since Lumiere still has trade-offs with Fever, the following question remains:

▶ **Question 1.** *Does there exists a view synchronisation protocol for the partial synchrony model that achieves the same efficiency bounds as Fever, but which can accommodate unbounded clock drift prior to GST?*

References

1. Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069, 2018. arXiv:1803.05069.
2. Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 337–346, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331612.
3. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001. doi:10.1007/3-540-45682-1_30.
4. 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.
5. Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. *Distrib. Comput.*, 35(6):503–532, sep 2022. doi:10.1007/S00446-022-00432-Y.
6. Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making byzantine consensus live. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 23:1–23:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.DISC.2020.23.
7. Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, 2016. URL: http://hdl.handle.net/10214/9769.
8. Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017. arXiv:1710.09437.
9. Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine consensus is $\Theta(n^2)$: The dolev-reischuk bound is tight even in partial synchrony! In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 14:1–14:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.14.
10. Shir Cohen, Idit Keidar, and Oded Naor. Byzantine agreement with less communication: Recent advances. *ACM SIGACT News*, 52(1):71–80, 2021. doi:10.1145/3457588.3457600.
11. Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985. doi:10.1145/2455.214112.
12. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988. doi:10.1145/42282.42283.
13. 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.
14. Andrew Lewis-Pye. Quadratic worst-case message complexity for state machine replication in the partial synchrony model. *CoRR*, abs/2201.01107, 2022. arXiv:2201.01107.
15. Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems*, 1(2), October 22 2021. URL: https://cryptoeconomicsystems.pubpub.org/pub/naor-cogsworth-synchronization.

16 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine SMR. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 26:1–26:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.DISC.2020.26.

17 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000. doi:10.1007/3-540-45539-6_15.

18 Alexander Spiegelman. In search for an optimal 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 38:1–38:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.38.

19 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.

A Appendix

The following fairly nuanced observation regarding Cogsworth and Naor-Keidar came out of a private conversation with some of the authors of those papers. For the purpose of this discussion, we assume some familiarity with both of those papers [15, 16].

In both Cogsworth and Naor-Keidar, there is a certain known bound, which we will call δ^* here, and which determines the delay before a correct processor gives up on one relay and tries the next. In those papers, δ^* is set to equal 2Δ, which gives the results described in Table 1. By playing with different values for δ^*, however, it is possible to achieve a trade-off between worst-case latency and expected complexity. While the corresponding results do not significantly impact the narrative of this paper, they may be of interest to the dedicated reader.

In the following discussion, we will consider only the case of a static adversary. We assume that δ^* may be much smaller than Δ, but is at most 2Δ. For Cogsworth, the possibility that δ^* may be much smaller than Δ now means that the worst-case latency is $O(f^2\delta^* + f\Delta + \delta)$, while the worst-case complexity remains $O(fn^2 + n)$. Expected latency is $O(\Delta)$ (and $O(\delta)$ if $f = 0$).

The issue with making δ^* very small is that, while this decreases the worst-case latency, it increases the *expected* complexity in the case that $\delta > \delta^*$. In this case, the expected complexity becomes $O(n^2)$ even with benevolent faults (the standard version of Cogsworth has expected complexity $O(n)$ in the case of benevolent faults).

For Naor-Keidar, if actual latency is at most δ^*, then worst-case latency is $O(f^2\delta^* + f\Delta + \delta)$ and worst-case communication is $O(f^2n + n)$. Expected latency is $O(\Delta)$ (and $O(\delta)$ if $f = 0$). The expected communication cost is $O(n)$. If actual latency is $> \delta^*$, then the worst-case latency is $O(f^2\delta^* + f\Delta + \delta)$ and worst-case communication is $O(n^2f + n)$. Expected latency in this case is $O(\Delta)$ and expected communication is $O(n^2)$. So, again, there is a trade-off. Setting a small value of δ^* decreases the worst-case latency, but increases the expected communication in the case that $\delta > \delta^*$.

Nova: Safe Off-Heap Memory Allocation and Reclamation

Ramy Fakhoury
Technion, Haifa, Israel

Anastasia Braginsky[1]
Red Hat Research, Ra'anana, Israel

Idit Keidar
Technion, Haifa, Israel

Yoav Zuriel
Technion, Haifa, Israel

Abstract

In recent years, we begin to see Java-based systems embrace off-heap allocation for their big data demands. As of today, these system rely on simple ad-hoc garbage-collection solutions, which restrict the usage of off-heap data. This paper introduces the abstraction of *safe off-heap memory allocation and reclamation (SOMAR)*, a thread-safe memory allocation and reclamation scheme for off-heap data in otherwise managed environments. SOMAR allows multi-threaded Java programs to use off-heap memory seamlessly. To realize this abstraction, we present Nova, *Novel Off-heap Versioned Allocator*, a lock-free SOMAR implementation. Our experiments show that Nova can be used to store off-heap data in Java data structures with better performance than ones managed by Java's automatic GC. We further integrate Nova into the open-source Oak concurrent map library, which allows Oak to reclaim keys while the data structure is being accessed.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases memory reclamation, concurrency, performance, off-heap allocation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.15

Supplementary Material *Software (nova open source code)*:
`https://github.com/li0nr/Nova-Safe-off-heap-memory-allocation-and-reclamation/`
archived at `swh:1:dir:f756390e070c9f32bbbfee2e514cdc5f141a591b`

1 Introduction

Programmers nowadays are accustomed to managed-memory environments as provided by today's most popular programming languages [12]. In such environments, memory is allocated from a region called *heap*, which is managed by the system, and is reclaimed by an automatic *garbage collection (GC)* algorithm. This paradigm is attractive as it reduces programming complexity, bugs, and memory leaks.

Today's leading GC solutions still struggle to scale with the volume of on-heap memory, especially in memory-hungry "big data" systems.

Figure 1 shows the throughput of Java's ConcurrentSkipListMap with a read-write workload (25% put, 25% delete, 50% get), where values are on heap managed by Java17's state-of-the-art GC solutions (star-square, diamond, square and circle shapes), compared to off-heap values managed by our solution (hexagram). Each experiment utilizes 16 threads. Of the machine's 128GB of DRAM, 110GB are allocated to Java's heap in the on-heap experiments, whereas in the off-heap experiment, 80GB are allocated off-heap and 30GB

[1] Work done before joining Red Hat Research.

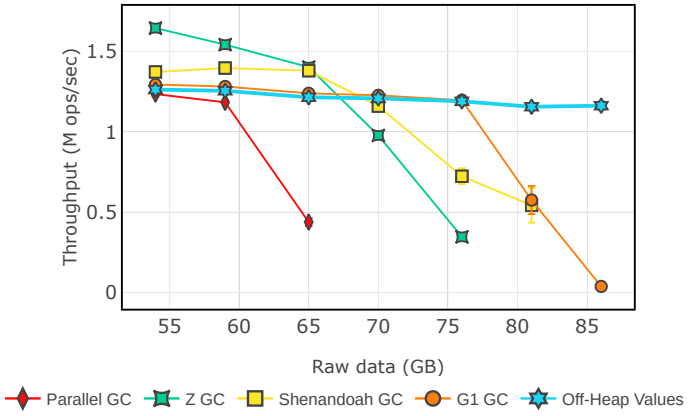


Figure 1 Performance of Java's ConcurrentSkipListMap with state-of-the-art Java17 GC implementations versus off-heap allocation managed by Nova.

on-heap (for more details see Section 4). The only on-heap GC implementation that can sustain 80GB of raw data is G1, and its performance degrades as the amount of ingested data increases, whereas our off-heap solution continues to perform well under memory duress. This conundrum has led Oracle to offer an off-heap allocation API in JDK as of version 14 [7, 22]. And indeed, a number of systems developed in Java now employ off-heap memory buffers alongside on-heap objects. For example, off-heap buffers are used to store in-memory tables in databases like Cassandra [14] and HBase [4, 20, 25]. Another example is the Druid analytics database, which uses off-heap buffers as temporary space for processing queries (e.g., large table merges) [13]. Nevertheless, these systems use the off-heap memory only in specific use cases where GC is straightforward. In particular, the off-heap buffer is reclaimed all at once – when an entire memory table is flushed to disk or when the query computation completes – and, by design, it is assured that all the threads that might have accessed the buffer have terminated. In essence, the off-heap memory usage in these use cases is *grow-only* as long as it might be referenced.

Reclaiming memory while data is still being accessed is challenging: Consider a memory location released by some thread in a system with concurrent access – we must prevent a situation where it is reclaimed and re-used while it is still being accessed by another thread.

The recently-developed Oak key-value map [27] pushes the use of off-heap allocation further: Oak is an open-source Java library that allows programmers to store keys and values off-heap, while managing meta-data on-heap. Oak currently uses a simple grow-only (release-at-once) memory manager for keys and a naïve lock-based memory manager for values, which locks off-heap objects on every access.

In this paper, we take off-heap memory management another step forward. In Section 2, we introduce the abstraction of *safe off-heap memory allocation and reclamation (SOMAR)*. Its API supports allocation, read/write access of allocated data, and deletion of previously allocated data. A SOMAR service reclaims and re-allocates off-heap data in a manner that does not restrict concurrent access: User code may freely copy off-heap objects and hold them for unknown periods. For example, multiple threads might retrieve the same data element from a map and then use it in lengthy computations. Similarly, an iterator might spend some time processing a map element before proceeding to the next one. Java threads do not generally "release" or "close" an object or iterator once it is no longer in use, and so a SOMAR implementation has no way of tracking the allocated objects' accurate reference

counts. Thus, depending on reference count for GC is not an option. A key challenge a SOMAR solution needs to address is ensuring *safety*, namely, detecting every unsafe attempt to access reclaimed (and re-used) memory. Providing such safe access facilitates a familiar programming experience in a managed environment.

In Section 3 we present *Nova*, Novel Off-heap Versioned Allocator, a lock-free SOMAR implementation. Nova may be used in lock-free data structures, as well as lock-based ones. In addition to safety and lock-freedom, Nova ensures *robustness* [34]– a stalled thread delays the reclamation of at most one data item. We keep the design simple so it is easy to implement and understand. In a nutshell, Nova uses version numbers to detect unsafe access. We use optimistic access [9] for read and a variant of hazard pointers [28] for writes.

We implement Nova in Java and use it to manage off-heap data indexed via on-heap data structures. Though no previous work provides the SOMAR functionality, we compare Nova to alternative SOMAR implementations using known reclamation approaches [15, 30] as well as to on-heap data structures managed by Java's automatic GC. Our results, reported in Section 4, show that SOMAR solutions outperform Java's GCs in all tested workloads. We further show that when primitive types are used to reference off-heap data (instead of Java objects), Nova outperforms all alternative SOMAR implementations, and otherwise performs similarly to them.

We further integrate Nova into the Oak open-source library and compare it to Oak's original grow-only and lock-based reclamation schemes. Our results show that the lock-based solution, when used also for keys, does not scale with more than one thread, whereas the Nova-based Oak scales linearly with the number of threads. The Nova-based Oak outperforms Java's ConcurrentSkipListMap and in fact performs as well as Oak with grow-only memory management (in which memory is not reclaimed at all).

It is important to note that SOMAR is different from *safe memory reclamation* (SMR) for C/C++ data structure libraries, which gained substantiated attention in recent years [3, 5, 6, 8, 9, 15, 16, 21, 23, 28–30, 32, 34] this because SMR assumes that the scope of a reference is limited to an explicitly protected part of the code, for example, the execution of the data structure operation. Thus, SMR solutions do not allow programs to *externally* access the memory they manage from outside the data structure library. This means, for example, that a map's get operation must return a copy rather than a reference to a data item that resides in the map. Additionally, iterators are not supported, as these continue to reference the data structure between calls. In contrast, SOMAR allows references to be used externally and to be copied freely. In Section 5, we go over related work in more detail.

In summary, we provide efficient memory management for off-heap data that can be used outside the scope of a given data structure, a functionality that is missing in today's managed environments. Section 6 concludes our paper. Correctness proofs are given in Appendix A. Appendix B includes an extension of Nova that mitigates fragmentation.

2 SOMAR

We introduce the abstraction of safe off-heap memory allocation and reclamation, SOMAR. SOMAR is intended for managed environments that support allocating unmanaged *off-heap* memory. Such allocation is natively supported in Java versions 14 and up. In new Java versions, it is possible to allocate off-heap memory using *memory segments* [22]. Such memory is not subject to GC and its access is not safe against races between memory access and delete-reuse.

SOMAR allows applications in a managed environment to allocate, use, and de-allocate off-heap memory in a safe manner. The allocated unit is called a *slice*. Applications can use their allocated slices for storing data, for instance, the keys and values of a key-value map. An application may invoke a slice delete operation, allowing SOMAR to reclaim its memory for reuse in future allocations.

Slices are accessed via on-heap *safe-pointers*. Applications can store safe-pointers in data structures as well as in ephemeral objects and can copy them without informing SOMAR. Thus, multiple safe-pointers may address the same slice, and the service has no way of counting how many safe-pointers reference a given slice. User code might invoke a delete operation on a slice via one safe-pointer, while additional safe-pointers continue to reference the now deleted slice. Importantly, SOMAR ensures safety in case the slice's memory is reused: all attempts to access slices that have been deleted and reallocated result in errors. In other words, SOMAR protects against access-delete-reuse races.

Reading and writing slice data is done using user-defined functions (lambdas) that operate directly on off-heap memory. We assume that the lambda functions have no side effects beyond accessing the slice, and the function passed to read operations does not update the underlying memory, moreover we assume that lambdas do not fail upon reading illegal values (e.g., infinite loop, dividing by zero). For simplicity, we assume that there are no nested calls to the safe-pointer from within lambda functions, though it is not difficult to relax this assumption, as explained below.

SOMAR supports the following API:

- *safe-pointer allocateSlice(length)* allocates a slice of the requested length;
- *buf read(safe-pointer ptr, lambda f())* allows the application to read the slice using f;
- *buf write(safe-pointer ptr, lambda f(), param))* allows the application to write to the slice using f;
- *boolean delete(safe-pointer ptr)* de-allocates the slice associated with the safe-pointer.

The methods may fail upon attempting to access deleted data.

Privatization. Ensuring safety in the presence of potential concurrent deletions incurs a synchronization cost. *Privatization* [24, 33] is a technique for reducing this cost when it is known that a slice is not accessible by more than one thread. Consider, for example, a thread that allocates a new slice for inserting a new element into a shared data structure. The thread writes its data into the slice when it is still *private*, namely inaccessible to other threads. In this case, the write does not need to be protected from concurrent deletions of the same slice. Similarly, if a thread allocates a slice and then fails to insert it into a shared data structure and deletes it, the delete occurs when the slice is still private and so concurrency races are not possible. To support unprotected operations for private slices, we add the following methods to the API:

- *writePrivate(safe-pointer ptr, lambda f(), param)* ;
- *deletePrivate(safe-pointer ptr)* ;

It is the responsibility of the programmer to call the private methods only when it is ensured that the slice is private.

3 Algorithm

Nova is an algorithm for implementing SOMAR. We give a high level view of Nova in Section 3.1 and describe its data layout in Section 3.2. In Section 3.3, we present the algorithm's pseudocode, while abstracting away the memory model. Implementing the algorithm in weak-memory models is discussed in Section 3.4. Correctness is discussed in Appendix A.

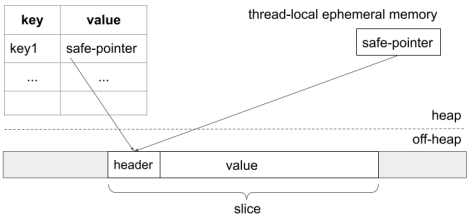


Figure 2 Example: A Nova slice with two safe-pointers.

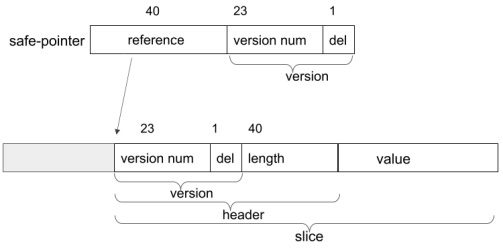


Figure 3 An example layout of Nova headers and safe-pointers.

3.1 Overview

As an implementation of SOMAR, Nova allocates slices within a large memory area, which it requests from the OS via the JDK and never released. It uses on-heap safe-pointers to refer to slices. User code can copy these safe-pointers freely without informing Nova, and might delete a slice when there exist additional safe-pointers addressing it. A key challenge we address, therefore, is detecting access to obsolete slices via old safe-pointers. Nova must ensure that every access via a safe-pointer to a slice that had been deleted and subsequently reallocated results in a failure. In other words, it must prevent access to the memory location that used to pertain to that slice before its deletion and now pertains to another. To this end, each slice has an off-heap *header*, which is used to track its validity as we explain next. A slice's header is unique while multiple on-heap safe-pointers may refer to the same header.

For simplicity, we associate each slice with a static header. Whenever a slice is re-used, the header's location does not change. We place the header right before the slice. This approach is simple, efficient, and avoids indirection. However, it also restricts future allocations and may thus lead to fragmentation when using variable allocation sizes. In many cases, this can be a reasonable compromise: It is cheaper than automatic GC, which relocates data to avoid fragmentation. And it allows released memory to be reused whenever allocation sizes recur, for example, in a map with fixed-size keys. This is in contrast to existing Java-based systems that use grow-only off-heap memory without reusing *any* memory as long as some allocated data might be referenced [4, 13, 14, 20, 25]. For applications with frequent allocation and de-allocation with variable allocation sizes, we present in the Appendix B an approach for mitigating fragmentation at the cost of a fixed memory overhead.

Figure 2 illustrates a Nova slice allocated within a block and two safe-pointers pointing to the slice's header. The slice holds a value stored in a key-value map, and the safe-pointer depicted on the left is an entry in the key-value map. The safe-pointer on the right is ephemeral – it has been retrieved from the map by some thread.

The header includes a *deleted* bit, which is set when the slice is deleted, and the slice's *length*. The deleted bit is also stored in the safe-pointer. Every attempt to access the slice checks the deleted bit in the safe-pointer and fails if it is set. As long as a deleted slice is not re-used, these bits suffice for checking its validity, because the slice remains deleted and so the deleted bit remains set.

To discover that a deleted slice has been reallocated, we employ *versions*, which are stored in both headers and safe-pointers, and change whenever slices are reallocated. Attempts to access the slice verify that the versions in the safe-pointer and in the header are the same.

While the header and safe-pointer suffice to ensure safe access in a sequential execution, an additional mechanism is needed to ensure correct concurrent access. Consider a slice deleted by some thread – we must prevent a situation whereby the slice is reclaimed and re-used while it is still being accessed by another thread.

One simple way to address such concurrency races is using a per-slice read-write lock. We forgo this approach for two reasons. First, we have designed Nova to be lock-free so that it may be used as a building block in lock-free applications, e.g., concurrent lock-free data-structures. Second, using locks can degrade performance, as we show in Section 4 when comparing a Nova-based implementation of Oak to Oak's lock-based reclamation mechanism.

Instead, we opt for a lock-free solution using a combination of *optimistic reads* [9] and *hazard pointers* [28] for writes, whereby writing threads announce which slices they are accessing to prevent these slices from being reclaimed. As every access to a slice's data is via a safe-pointer, the scope of each slice access is limited to the execution of one safe-pointer method. Our assumption that safe-pointer calls are not nested allows us to keep one hazard pointer per writing thread. (It is straightforward to extend the solution to allow nesting of safe-pointer calls by keeping multiple hazard pointers.) For reads, we improve performance by eliminating hazard pointers altogether. Rather, reads are allowed to temporarily observe illegal values, but such situation are detected (via the version) and result in failure.

3.2 Data structures and layout

Both the safe-pointer and the header hold a version, which includes a version number *num* and a *deleted* bit. Versions are assigned using a global monotonically (infrequently) increasing *NovaEra* counter, an atomic integer initialized to 1. For simplicity, we assume that the counter does not wrap around. In practice, it is acceptable for it to wrap around as long as it takes sufficiently long to repeat the same counter value so that no safe-pointers holding the version used in its previous incarnation exist in the system. Slices are addressed via *references*. Each safe-pointer holds a version and a *reference*. An example safe-pointer layout is illustrated in Figure 3. In this example, the reference consists of 40 bits, which can address up to 1TB of memory.

In our implementation, the entire safe-pointer occupies a single (long) memory word, and so can be stored in a primitive long type rather than a Java object. Furthermore, its reference and version may be updated atomically together via a single CAS instruction. Using bigger safe-pointers is possible, but would degrade performance as (1) they would need to be stored in Java objects; and (2) since as of today, Java does not support multi-word CAS, we would need to use a user-level multi-word CAS solution [18]. Note that the length field in the header cannot be larger than the offset, and therefore the header also fits into one memory word that can be updated using a single CAS instruction.

Because multiple safe-pointers may reference the same slice, the off-heap header is always the source of truth regarding the version and deletion state. To this end, on slice deletion, the header is updated before the safe-pointer.

Nova uses a number of shared data structures, as summarized in Table 1. To manage hazard pointers, it holds a dedicated *Thread Array of Pointers (TAP)*, with references to slices accessed by all ongoing writes. The TAP has one slot per thread, holding a reference to the slice that the thread is writing to (if any).

New slices are allocated either from unused memory or from a *free list* holding available (free) slices. The free list is managed on-heap, as a skiplist sorted by size. When a slice is deleted, it is not immediately added to the free list. This is because the slice should not be reallocated as long as it might be written by a concurrent thread. Instead, a deleted slice is temporarily added to a *release list*, which is managed as an on-heap array. To avoid contention, each thread manages its own release list, and infrequently (e.g., after adding 1000 items to the release list) promotes eligible slices from its release list to the shared free list. To ensure robustness, threads also periodically (but infrequently) check if there are other

Table 1 Shared data structures used by Nova.

Data Structure	Access Pattern
NovaEra	shared, updated using atomic increment
TAP	each entry written by one, read by all
free list	shared by all threads
release lists	each owned by one thread, accessible to helping threads

threads whose release lists are full, and if so, *help* them by promoting items from their release list to the free list. Before adding a slice to the free list, the de-allocation process verifies that no TAP entries refer to it. In addition, to ensure that reallocation uses a different version number, the NovaEra is incremented before slices are migrated to the free list. We increment NovaEra using an atomic increment instruction to address races among concurrent threads.

3.3 Nova algorithm

We now detail how Nova's API is supported. For brevity, we describe the algorithm assuming a sequentially consistent memory model, and defer the discussion on its correctness in other memory models to Section 3.4.

Algorithm 1 Nova safe-pointer read and write methods.

```
1:  procedure READ(ptr, f)
2:      ⟨ver, slice, header⟩ ← locateSlice(ptr)
3:      if slice = ⊥ then
4:          return ⊥
5:      ret ← f(slice)                                   ▷ read from slice
6:      if header.ver ≠ ver then                          ▷ validate
7:          return ⊥
8:      return ret
9:  procedure WRITE(ptr, f, param)
10:     set hazard pointer in TAP to ptr
11:     ⟨ver, slice, header⟩ ← locateSlice(ptr)
12:     if slice = ⊥ ∨ header.ver ≠ ver then
13:         set hazard pointer in TAP to ⊥
14:         return ⊥
15:     ret ← f(slice, param)                             ▷ write to slice
16:     set hazard pointer in TAP to ⊥
17:     return ret
18: procedure LOCATESLICE(ptr)          ▷ returns ⟨version, slice, header⟩ triple.
    ▷ version includes version num and deleted bit
19:     myVer ← ptr.ver
20:     if myVer.deleted then return ⟨⊥, ⊥, ⊥⟩
21:     header ← address referred to by ptr.ref
22:     slice ← address of header's slice (header + 8)
23:     return ⟨myVer, slice, header⟩
```

Reading and writing. Pseudocode for Nova's reads and writes appears in Algorithm 1. Both operations use the safe-pointer's locateSlice method in order to extract from the safe-pointer the reference to the accessed slice and its header, as well as its version. Reading and writing occur, via the user-provided lambda function f, and return value is the user-defined return value of f.

In addition, every operation performs *validation* by comparing the version in the safe-pointer to the one in the header (along with the deleted bit) and returns failure in case the version is no longer valid. To allow NovaCounter to wrap-around, we only check whether the versions are equal (As noted above, we assume that the cycle is long enough to ensure that by the time a version is re-used, no obsolete safe-pointers holding its value from previous incarnations exist). Reads are optimistic, performing validation after the actual read (Line 6), whereas writes perform validation before the actual write (Line 12).

In both cases, if the versions (and deleted bits) do not match, it means that the slice has been deleted (and possibly subsequently reallocated), and the operation fails. As an optimization – to expedite future operations, mismatch detection – it is possible to mark the safe-pointer's version as deleted in such cases (using a CAS). This optimization is omitted from the code as it has no effect on correctness.

To synchronize with concurrent deletion and allocation operations, writes proceed through TAP: a writing thread first writes the accessed reference to its TAP entry. As long as the write is ongoing, the TAP entry holds its reference, ensuring that the slice is not retired into the free list. When writing is done, the writing thread sets its TAP entry to $\perp$, allowing the slice to be garbage collected (this write can be lazy because it is not required for correctness).

To make reads lightweight, we allow them to proceed without accessing TAP. Thus, nothing prevents a read slice from being deleted and reallocated before the read operation ends. This means that validation must occur after the value is read; ensuring this execution order in a weak memory architecture is discussed in the next section.

The pseudocode of the privatized write method is not detailed, as it simply applies the given lambda function to the slice.

Slice allocation and deletions. Algorithm 2 describes Nova's allocation and deletion methods. A new slice is obtained either from unused memory or from the free list. We assume that slices are obtained atomically, namely, each slice is assigned to a single thread. We initialize the new slice's header with a version num from the current NovaEra, the deleted bit unset (Line 28), and the slices's length. The off-heap header is the source of truth regarding the slice's allocation, and so once it is set, the slice is considered as allocated. The method returns a safe-pointer with a reference to the slice's header and the new version.

After locating the slice, a delete operation updates the delete bit in the header and the safe-pointer to true. The header is updated first, using a CAS, so all races with concurrent deletions are resolved by the order in which the CASes are scheduled (only the first succeeds). Note that in case delete is called for an already deleted slice, the CAS keeps it unchanged, and delete returns false. If the CAS succeeds, we add the deleted slice to the thread's release list. Either way, the safe-pointer's delete bit is set. The privatized delete method is the same except in that it updates the header without a CAS and always succeeds. If additional safe-pointers refer to the same slice, the deleted bits in these safe-pointers remain unset, but future attempts to access the slice via these safe-pointers detect the mismatch: reads concurrent with a deletions detect the header change when checking the version at the end of the read. While, in the case of a write being concurrent with a deletion, the slice is registered in the writer's TAP, preventing the slice's garbage collection. Nova does not determine the

Algorithm 2 Nova's allocation and deletion methods.

```
24: procedure ALLOCATESLICE(length)
    ▷ atomically get new slice of size length
25:     newSlice ← get reference to new slice header
26:     newPtr ← new safe-pointer
27:     newPtr.ref ← newSlice
28:     newPtr.ver ← ⟨NovaEra, false⟩
29:     newSlice.ver ← newPtr.ver
30:     newSlice.length ← length
31:     return newPtr
32: procedure DELETE(ptr)
33:     ⟨ver, slice, header⟩ ← locateSlice(ptr)
34:     if slice = ⊥ then return false
35:     len ← header.len
36:     flag ← CAS ( ⟨header.ver, header.length⟩, ⟨ver, len⟩,
                     ⟨⟨ver.num, true⟩, len⟩ )
37:     ptr.ver.delete ← true
38:     if flag then
39:         add slice reference to (thread-local) release list
40:     return flag
41: procedure RETIRE
42:     atomically increment NovaCounter
43:     for all ref in (thread-local) release list do
44:         if no TAP entry contains ref then
45:             append ref to (shared) free list
46:             remove ref from release list
```

execution order between concurrent deletes and writes of the same slice. Rather, it ensures that a deleted slice is not reallocated as long as ongoing writes can access it and that no future reads or writes that begin after the deletion is complete successfully access the slice.

Each thread periodically calls the retire procedure (Algorithm 2, Lines 41–46) to move slices from its release list to the shared free list. It increases the NovaVersion and then migrates all slices that have no pending accesses registered in TAP. This ensures that whenever a slice is reallocated, it is assigned a different version than it had before. We omit a helping mechanism from the pseudocode, whereby threads retire eligible slices from other threads' full release lists to achieve robustness.

3.4 Synchronization over weak memory models

For simplicity, we presented Nova's pseudocode above assuming the memory is sequentially consistent. We now identify the points in our pseudocode where synchronization instructions must be used for correct execution on x86-TSO. To ensure that a read operation checks the version *after* reading the data, it issues a load fence after Line 5. All other steps of the read operation can be safely reordered. A write must guarantee that the TAP entry is set before the version check that precedes the actual write. This is ensured by adding a full fence after the TAP update in Line 10. In addition, a store fence is required after Line 15, before setting the TAP entry to $\perp$, to ensure that the $\perp$ is not visible before the write.

A delete operation needs to set the delete bit in the off-heap slice before modifying the safe-pointer. This update is already done using a CAS instruction in order to deal with potential races among deleting threads Line 36. The retire procedure must ensure that the incremented NovaEra is visible before adding slices to the free list, so that older versions will not be used in new allocations of these slices. This is ensured by using an atomic increment instruction to update the counter.

4 Evaluation

We now evaluate Nova and two other SOMAR implementations. In Section 4.1 we use SOMAR to manage off-heap data organized in Java data structures and in Section 4.2 we integrate Nova into the Oak library.

Experiment setup. All the code is written in Java 17 and built with Apache Maven 3.8. In Section 1, we compared four state-of-the-art Java GC solutions. Here, we experiment with – G1, which is best under memory stress according to our experiments, along with the latest GC releases ZGC and Shenandoah, which show the best performance when ample memory is available, in our experiments. We used a customized synchrobench tool [17]. For each data point, we start a new JVM, warm it up for one test, and then repeat the test three times and plot the average.

Experiments were run on an AWS instance c5ad.16xlarge, AMD EPYC 7R32 with 128GB RAM. To check whether there were any platform-specific effects, we repeated some of the experiments on ARM architecture. The trends were similar and the results presented in the Appendix C. All experiments utilize 32 cores (without hyper-threading) on one NUMA node and the OS is Ubuntu 20.04.

4.1 Off-heap data indexed by Java data structures

Benchmarks. Our experiments highlight two different scenarios where off-heap allocation can be beneficial. The first is a user-defined data structure that can store primitive types, reducing GC overhead and eliminating a level of indirection in referencing actual data. The second is a big data scenario, where the system runs under memory stress. Specifically, we use SOMAR to manage off-heap data organized in the following on-heap Java data structures:

LL-map A key-value map using Harris's linked list [19] with on-heap nodes referencing keys and values off-heap (using safe-pointers or Java objects).

CSLM Java's ConcurrentSkipListMap, where keys are on-heap, and values are Java objects referencing off-heap slices. (CSLM does not support primitive types.)

In the LL-map, a put overwrites the slice data if the key is present, and creates a new slice otherwise. In the latter case, the privatized write is used. If the insertion fails (due to a race), the slice is then deleted using the privatized delete method. Because CSLM values are Java objects, every put allocates a new slice and safe-pointer and deletes the old one if it exists. Hence here, all writes are privatized.

In both data structures, we experiment with write-heavy (50% put 50% delete), read-heavy (5% put 5% delete 90% get), and read-write (25% put 25% delete 50% get) workloads. Each test lasts 30 seconds.

Because the linked list has a linear search time, the LL-map size cannot scale, so we keep it to ∼100MB. We initialize it with ∼65K entries holding 512 byte keys and 1KB values. We experimented with other key and value sizes (both larger and smaller), and the trends were similar. Keys are drawn uniformly at random from a sub-range consisting of ∼130K keys, to allow roughly 50% of the deletions to succeed and 50% of the insertions to insert new values.

In order to keep the CSLM experiments tractable, we run them with ∼10GB of raw data and scale down the available RAM budget accordingly. To this end, the CSLM is initialized with 10 million entries holding 128 byte keys and 1KB values (Here too, experiments with different key and value sizes with the number of entries scaled to consume the same overall memory size showed similar trends). Keys are drawn from a sub-range consisting of 20 million keys. The total RAM budget is 14GB. For off-heap (SOMAR) solutions, this is split into 4GB on-heap and 10GB off-heap. Since the LL-map size does not scale, this data structure is not intended for use under memory stress. Hence, we do not limit the available memory in LL-map experiments.

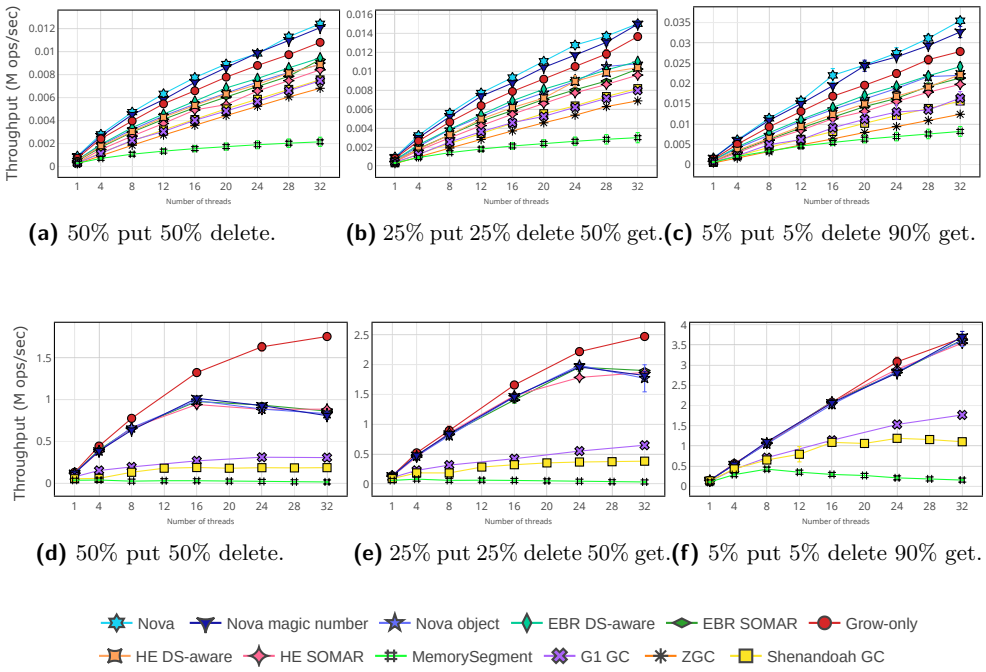
Compared solutions. While no previous work has implemented SOMAR, we can implement a similar service by adapting known safe memory reclamation techniques to work in our managed environment. We chose to compare Nova to *Epoch-Based Reclamation (EBR)* [15] and *Hazard Eras (HE)* [30]. We also experimented with Interval-Based Reclamation [34] in some workloads and got similar results to EBR, so we refrained from a detailed study of this approach.

To use EBR and HE in our context, we need to reference each piece of off-heap data (key/value) with an on-heap representation similar to Nova's safe-pointer. Whereas the safe-pointer can be implemented as a primitive long, the data required by EBR and HE does not fit into one word, and so we use Java objects. As noted above, in the CSLM experiment we must use Java objects also for Nova's safe-pointers. To evaluate the impact of using objects, we run the LL-map experiments with Nova safe-pointers implemented both as primitive longs and as objects.

Additionally, we need to explicitly specify the scope of the safe access, i.e., to wrap all accesses to off-heap data with explicit *protect* and *un-protect* calls. We do this in two ways: (1) The SOMAR approach, where the off-heap representation wraps each access to an off-heap data item, in order to comply with our API. This way, at every node in a linked list traversal, we call protect before reading the key and un-protect afterward. (2) A *data-structure-aware (DS-aware)* approach, where protect is called at the beginning of the data structure operation and un-protect is called at the end. In the latter case, the protected region spans the entire linked list traversal, which accesses all keys along the way. For Nova, we use only the SOMAR approach, as Nova was explicitly designed for SOMAR. Finally, we implement a variant of Nova that reduces fragmentation at the cost of some memory overhead; this technique is called *magic numbers*, and for space limitations, is deferred to Appendix B.

We compare the three SOMAR approaches (Nova, EBR, and HE) to five baselines: (1) on-heap allocation with Java17's default G1 GC; on-heap allocation with Java17's newly introduced (2) ZGC; (3) Shenandoah GC; (4) raw Java memory segments [22] for fine-grain allocations; and (5) grow-only off-heap memory without reclamation with safe-pointers implemented as Java objects holding the length and offset of the off-heap data.

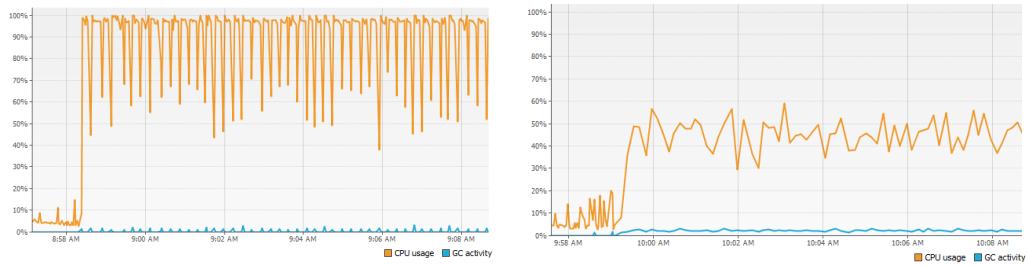
LL-map results. Figure 4(a)–(c) depicts the throughput results for the LL-map. We see that all SOMAR solutions outperform Java's GC solutions and memory segments across all workloads. For memory segments, this is expected since they are not designed for fine-grain allocation scenarios. In these experiments, memory stress is not a factor, and GC activity is a performance bottleneck. We presume that this occurs due to the larger heap space that Java needs to manage. Using a profiler, we learned that the GC (fully on-heap) solution has 7 time more GC activity than Nova and 3x the allocation-churn rate of Nova. Moreover, Nova has the lowest GC time and the lowest allocation churn rate among all off-heap solutions.



(a) 50% put 50% delete. **(b)** 25% put 25% delete 50% get. **(c)** 5% put 5% delete 90% get.

(d) 50% put 50% delete. **(e)** 25% put 25% delete 50% get. **(f)** 5% put 5% delete 90% get.

Figure 4 LL-map (top-row) and CSLM (bottom-row) throughput (Mops/sec) versus number of threads.



(a) Nova CSLM with off-heap data. **(b)** On-heap Java CSLM.

Figure 5 CPU usage and GC activity in the CSLM experiment measured by VisualVM.

Nova achieves the best performance among SOMAR solutions. Its peak throughput is more than 2x that of the Java GC in all workloads, and it outperforms all other off-heap solutions by 20–30%. It even performs better than the grow-only solution that does not reclaim memory at all. Part of the benefit can be attributed to implementing the safe-pointer as a long, saving a level of indirection. When Nova uses object safe-pointers it performs similarly – or only slightly better than – alternative SOMAR solutions, and slightly (up to 10%) worse than the DS-aware HE and EBR implementations. The SOMAR solutions of EBR and HE perform worse than their DS-aware counterparts because they require many protect/un-protect calls during a list traversal whence many (off-heap) keys are read. Finally, magic numbers have a minimal performance impact.

CSLM results. The CSLM results are shown in Figure 4(d)–(f). In all these experiments, ZGC crashes with an OutOfMemoryError exception when allocated the same DRAM budget as other solutions and hence is not shown in the graphs. The Java on-heap GC solutions

Table 2 Memory consumption in GB, CSLM, read-write workload, 32 threads.

	Nova object	Nova magic	EBR	HE	memory segments	G1 GC
on-heap	2.8	2.8	2.9	3.0	3.2	13.0
off-heap	9.6	9.7	9.6	9.6	9.5	0

and memory segments fall far behind the off-heap solutions. At the same time, all SOMAR variants behave similarly (less than 1% difference). This is expected since the heavy work in a CSLM (searching) involves only keys, which are managed on-heap in all variants. By moving the (large) values off-heap, we reduce the load on the GC and improve overall performance. We also experimented with off-heap keys, but because we cannot store primitive types in a CSLM, this necessitated another level of indirection on every key access, deteriorating performance. This tradeoff between GC cost and indirection cost underscores the benefit of allowing programmers to use SOMAR alongside on-heap allocation in a managed environment. The cost of ensuring safe access is manifested in the difference between SOMAR and the grow-only solution, which is significant in updates and negligible in reads. Moreover, in the results shown in Figure 4, we can notice a large gap between the grow-only solution and the pure on-heap approaches, this gap depict the tradeoff between pure-on-heap approaches and leaky-off-heap approaches.

We use a profiler to assess the impact of GC. Figure 5 presents a timeline of CPU utilization and GC activity during the 32 thread experiment. Nova (Figure 5a) incurs periodic GC events (for managing the on-heap keys), which reduce its CPU utilization for short periods. In contrast, because this experiment runs under memory duress, the on-heap solution (Figure 5b) constantly engages in GC activity and its CPU utilization is constantly around 50%. This explains why the GC throughput is roughly half the off-heap solutions in this experiment.

Table 2 summarizes memory utilization of all solutions (except grow-only) in the read-write workload with 32 threads. We measure the memory consumption before the benchmark run (after initialization) and after it. The memory consumption of all algorithms was similar at both times. Note that GC requires slightly more total memory than the off-heap approaches, and that the overhead of Nova's magic numbers is small. Fragmentation does not occur in this experiment because all allocations are of the same size.

4.2 Nova in Oak

We integrated Nova into the Oak library [27] code.[2] Oak is designed to reference off-heap data using a primitive long, which we saw is beneficial in our LL-map experiment above. Because Oak stores primitive longs in the data structure, other SOMAR solutions are inapplicable. Oak currently offers two memory management options – (1) a grow-only manager that cannot reclaim memory concurrently with data access, and (2) a lock-based manager that uses locks to handle concurrency between data access and deletion.

We experiment with Oak using the following memory manager combinations:

Unreleased keys grow-only keys, lock-based values;
Grow-only grow-only for both keys and values;
Lock-based lock-based for both keys and values;

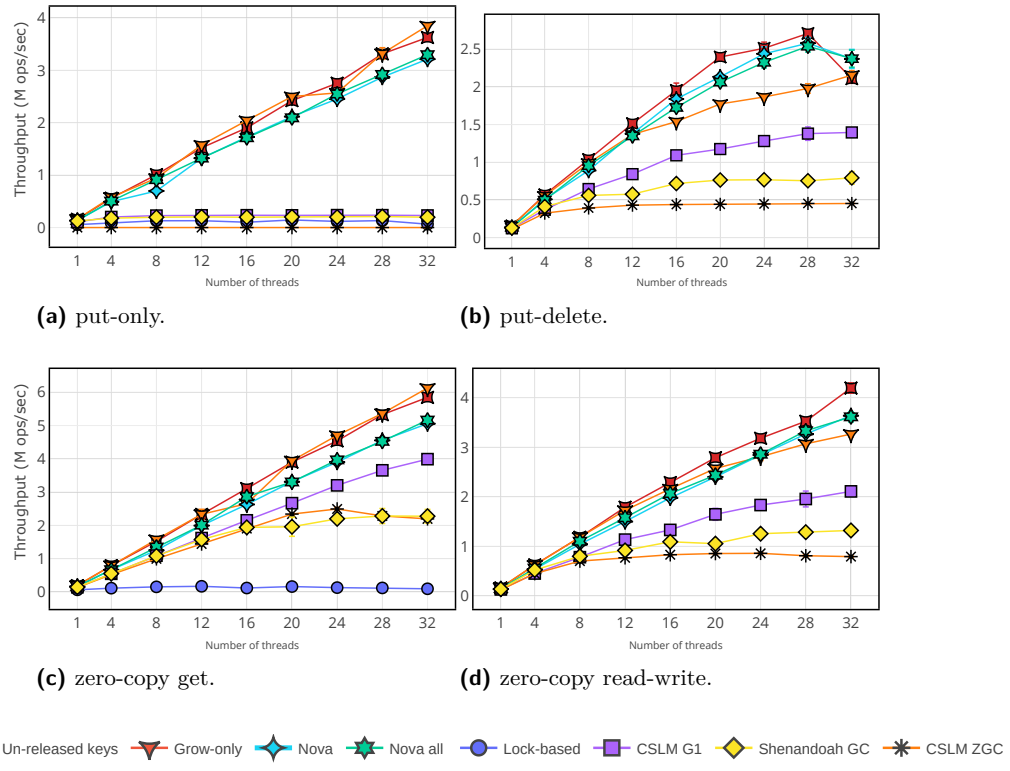
[2] `https://github.com/yahoo/Oak/pull/200`

Table 3 Memory consumption in GB, Oak read-write workload, 16 threads.

		Oak					CSLM	
		grow-only	unreleased keys	lock-based	Nova keys	Nova all	G1	ZGC
after init	on-heap	0.4	0.4	0.4	0.4	0.4	12.4	14.8
	off-heap	10.3	10.4	10.5	10.5	10.4	0	0
after run	on-heap	0.5	0.4	0.4	0.4	0.4	12.4	15.4
	off-heap	43.1	16.1	10.5	11.0	11.0	0	0

Nova keys Nova for keys, lock-based values;
Nova all Nova for keys and values.
We compare all of these to Java's CSLM, G1 GC, ZGC, and Shenandoah GC. All solutions run with a memory budget of 17GB. The off-heap solutions split this budget to 6GB on-heap and the rest off-heap. The grow-only variation has no memory restriction.

Our benchmarks generally follow the ones used in Oak's evaluation [27]. The map consists of 10 million key-value pairs with 100 byte keys and 1000 byte values, and the workloads are (a) put-only; (b) put-delete; (c) Oak's zero-copy get, where the application accesses the off-heap buffer directly using a lambda function; (d) a read-write workload with zero-copy gets. We ran the experiments fro 5 minutes, to demonstrate the effect of grow-only approach.



(a) put-only. **(b)** put-delete.

(c) zero-copy get. **(d)** zero-copy read-write.

Figure 6 Map throughput (Mops/sec): Oak with different reclamation schemes vs. CSLM.

The throughput results appear in Figure 6 and the memory consumption in Table 3. We see that the lock-based memory manager does not scale due to lock contention. Not surprisingly, the solutions that do not release keys (grow-only and unreleased-keys) are the fastest, since neither deals with concurrent access to keys where most of the data structure's work is done. Somewhat surprisingly, the remaining solutions, which have much smaller memory footprints, are no more than 7% slower than the grow-only solution. Moreover,

Nova performs similarly to grow-only in most workloads. We note that the small dip in performance in Figure 6b is duo to thread contention, as many threads compete to obtain a new slice from the allocator.

We see a small increase in Nova's memory usage over time, which stems from internal fragmentation in Oak data structures. We conclude that Nova offers a viable reclamation solution for Oak, allowing it to reclaim memory used for keys concurrently while accessing tha data-structure.

5 Related Work

SOMAR versus SMR. In recent years, many efforts have been dedicated to designing safe memory reclamation for concurrent programs in environments with manual garbage collection, most notably C/C++. Most of these works focus on concurrent *data structures (DSs)* [3, 5, 6, 8, 9, 15, 16, 21, 23, 28–30, 32, 34], and are designed to support safe deletion of DS elements. The main idea is to provide safe access guaranteeing that the accessed data is not concurrently reclaimed. For instance, in a linked listed traversal, protected access ensures that as long as the traversal holds a pointer to some node in the list, that node's memory is not reclaimed.

Typically, the scope of safe access is the DS operation, and data stored in the DS is externally inaccessible. Thus, retrieval operations cannot return pointers, and must instead copy the data, which is costly when data items are large. In addition, this scheme does not accommodate iterators, as they continue to hold a pointer into the DS after a getNext call returns. SOMAR does not limit the scope of the safe access, and instead allows memory to be reclaimed while there are still live references to it. In contrast SOMAR ensures that unsafe attempts to access data after it has been reclaimed fail. This approach is suitable for optimistic concurrency control, where computations may fail and need to be retried. Also, SOMAR is designed for a managed environment where only some data is off-heap, for instance, only the keys and values stored in a map and not the indexing pointers.

Techniques. Although the aforementioned works address a different problem, some do employ similar techniques. The main techniques used in memory reclamation are *reference counting*, *hazard pointers (HP)*, and *epoch-based reclamation (EBR)*. Modern schemes use different combinations of these, and some also rely on OS signals in order to allow progress in the presence of stalled threads.

Reference counting is used for lock-free reclamation in Hyaline [29], and is also used for automatic reclamation in OrcGC and DRC [2, 10]. As explained above, Nova is not amenable to reference counting because user code may copy safe-pointers without informing Nova.

Hazard pointers [11, 21, 28] and their variants [16] work as follows: before accessing data, a thread announces the pointer to the data is about to access, makes this announcement globally visible (using a CAS or fence), and then verifies that the pointer it is about to access has not been deleted from the data structure. The latter is tricky, requiring significant programmer effort and incurring high overhead (potentially a repeated DS traversal), as discussed in [6]. Nova's TAP entries are essentially HPs. Yet Nova eliminates the major difficulty (and source of overhead) in HPs as it does not rely on the DS to check whether the accessed slice had been deleted. Rather, we rely on versions (and the delete bit) to detect deletion. In addition, unlike data structures in manual GC environments, Nova HPs are required only for actual data updates and not for meta-data traversal (e.g., linked list pointers) and so one HP per thread suffices. This allows us to store a single TAP array with

entries for all threads, supporting efficient scans. Originally, HPs were used both for read and write access, but Nova follows the approach in Optimistic Access [8, 9, 31] and uses them only for writes while reads proceed optimistically.

Epoch-based reclamation [15, 26] was introduced in order to avoid synchronizing on every access as in HP. A number of variants of this approach exist in the literature, e.g., Debra/Debra+ [6], NBR/NBR+ [32], and ThreadScan [1]. Indeed, EBR has been shown to achieve better performance than HP by reducing the synchronization cost. Under this approach, threads share a global epoch and an announcements array. A thread wishing to access the data structure must declare its intent by publishing the current epoch in its entry in the announcements array. A block can be reclaimed if it was deleted from the DS before the lowest epoch in the announcements array. The disadvantage of EBR is that it delays reclamation and is not robust – a single stalled thread prevents reclaiming entire epochs (including memory it did not access). In some cases, OS signals are used to address such stalls [6, 32], which is not readily applicable in a JVM environment. Other solutions combine HP and EBR to improve EBR's robustness [3, 5, 23, 30, 34]. We looked into using epochs in Nova in order avoid a fence in the main execution path, but we found that in our setting, the performance penalty of fences was offset by the delay in reclamation introduced by epochs. Hence, we decided to forgo this solution.

6 Conclusions

The push for big data processing increases memory stress, while automatic GC solutions cannot always keep up with this growth. In recent years, we see a clear trend towards off-heap allocation of large chunks of data alongside the managed programming experience for other data. To facilitate this paradigm, we presented SOMAR, a safe memory allocation and reclamation scheme for off-heap data. SOMAR allows multi-threaded Java programs to use off-heap memory seamlessly, freely passing references to such data between threads. We presented Nova, an efficient lock-free SOMAR implementation. We used Nova for managing off-heap data organized in Java data structures, and showed that it performs better than fully on-heap solutions and state-of-the-art safe reclamation alternatives. We further integrated Nova into the open-source Oak concurrent map library, providing efficient support for reclamation of off-heap keys while the map is being accessed, a functionality that is not readily available today.

References

1 Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 123–132, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2755573.2755600.

2 Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. Concurrent deferred reference counting with constant-time overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 526–541, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454060.

3 Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2935764.2935790.

4 Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. Accordion: Better memory organization for LSM key-value stores. *PVLDB*, 11(12):1863–1875, 2018. doi:10.14778/3229863.3229873.

5 Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2486159.2486184.

6 Trevor Brown. Reclaiming memory for lock-free data structures: there has to be a better way, 2017. arXiv:1712.01044.

7 Maurizio Cimadamore. Bytebuffers are dead, long live bytebuffers! https://www.youtube.com/watch?v=RYrk4wvar6g&t=172s&ab_channel=FOSDEM, 2020.

8 Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *SIGPLAN Not.*, 50(10):260–279, oct 2015. doi:10.1145/2858965.2814298.

9 Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 254–263, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2755573.2755579.

10 Andreia Correia, Pedro Ramalhete, and Pascal Felber. Orcgc: Automatic lock-free memory reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 205–218, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437801.3441596.

11 Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 36–45, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2926697.2926699.

12 Larry Dignan. The most popular programming languages and where to learn them. https://www.zdnet.com/article/best-programming-language/, may 2021.

13 Docs Druid. Basic cluster tuning. https://druid.apache.org/docs/latest/operations/basic-cluster-tuning.html, retrived in April, 2022.

14 Jonathan Ellis. Off-heap memtables in cassandra 2.1. https://www.datastax.com/blog/heap-memtables-cassandra-21, 2014.

15 Keir Fraser. Practical lock-freedom. technical report ucam-cltr-579. In *Technical Report, UCAM-CL-TR-579 ISSN 1476-2986*, 2004. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

16 Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2009. doi:10.1109/TPDS.2008.167.

17 Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2688500.2688501.

18 Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare and swap, 2020. arXiv:2008.02527.

19 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, Berlin, Heidelberg, 2001. Springer-Verlag. doi:10.1007/3-540-45414-4_21.

20 Reference Guide Hbase. Hbase offheap read/write path. https://hbase.apache.org/book.html#offheap_read_write, retrived in April, 2022.

21 Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, may 2005. doi:10.1145/1062247.1062249.

22 Docs Java. Interface memorysegment. https://docs.oracle.com/en/java/javase/17/docs/api/jdk.incubator.foreign/jdk/incubator/foreign/MemorySegment.html, 2021.

23 Jeehoon Kang and Jaehwang Jung. A marriage of pointer- and epoch-based reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 314–328, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3385978.

24 Artem Khyzha, Hagit Attiya, Alexey Gotsman, and Noam Rinetzky. Safe privatization in transactional memory. *SIGPLAN Not.*, 53(1):233–245, feb 2018. doi:10.1145/3200691.3178505.

25 Yu Li, Yu Sun, Anoop Sam John, and Ramkrishna S Vasudevan. Offheap read-path in production the Alibaba story. https://blog.cloudera.com/blog/2017/03/, 2017.

26 Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, 1998.

27 Hagar Meir, Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Idit Keidar, Eran Meir, Gali Sheffi, and Yoav Zuriel. Oak: A scalable off-heap allocated key-value map. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 17–31, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3332466.3374526.

28 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, jun 2004. doi:10.1109/TPDS.2004.8.

29 Ruslan Nikolaev and Binoy Ravindran. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 987–1002, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454090.

30 Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras - non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 367–369, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3087556.3087588.

31 Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, pages 443–445, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3409964.3461817.

32 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: Neutralization based reclamation, 2020. arXiv:2012.14542.

33 Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 338–339, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1281100.1281161.

34 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 1–13, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3178487.3178488.

35 Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, feb 2008. USENIX Association. URL: https://www.usenix.org/conference/fast-08/avoiding-disk-bottleneck-data-domain-deduplication-file-system.

A Correctness

Nova guarantees the following properties:

Safety once a slice is retired, no operation writes to it or returns a value read from it.

Lock-freedom if threads attempt to perform Nova operations, eventually one succeeds.

Robustness a stalled thread prevents reclamation of at most one slice.

Lock-freedom is guaranteed because every Nova operation completes within a bounded number of its own steps, and fails only if another thread successfully deletes the same slice. Robustness is ensured because only slices referenced in TAP are prevented from being reclaimed, and helping is used to reclaim slices released by stalled threads.

We now discuss safety. First, we examine reads. If a slice is retired, its delete bit is set, so it has been deleted. If the delete occurs before the read begins, then the read either finds the delete bit set in the safe-pointer or a mismatch between the slice header version (which is either marked deleted or already reallocated with a new version) and the safe-pointer version. If the slice is deleted during or before the actual read in the lambda function in Algorithm 1, then the check after the read fails, and the read returns $\perp$. Otherwise, the slice is not retired before the read returns.

As for writes – if they begin after a slice is deleted, they fail in the same manner. If a deletion occurs simultaneously with the write, then the write may still succeed, but in this case, the slice will not be retired until the write completes. This is ensured because the write first sets the TAP entry and then verifies the slice's version against the safe-pointer. If this check succeeds, it means that the delete bit in the slice was not set before the writer's validation. And because a slice can be reclaimed by retire only after its delete bit is set, any attempt to reclaim the slice must begin after the write sets its TAP entry. Thus, the reclaim process sees the TAP entry set and does not reclaim the slice as long as the write is ongoing.

B Mitigating fragmentation

As noted above, we locate headers immediately before their respective slices within the same block. This provides locality of access and avoids the need for additional indirection. However, this restricts memory allocation because headers must remain static – they can be re-used when a released slice is reallocated, but must remain in the same physical location so that future accesses from old safe-pointers will detect the version change. This restriction prevents merging reclaimed slices and may therefore lead to fragmentation.

We note that it would have been possible to instead allocate headers in a dedicated headers block, but this would entail another level of indirection on every slice access so is less desirable. Similarly, it could have been possible to track occupied locations in a separate data structure (e.g., bitmap), but this would require a redundant access to the bitmap before every header access. We therefore forgo these options. Instead, we offer a probabilistic solution based on random *magic numbers* as we now discuss.

Our solution keeps headers located immediately before their slices but allows merging multiple freed slices to form a larger slice. When a slice is deleted, as before, the header remains in the same location with its deleted bit set while the slice is added to the free list. Allocation from the free list first attempts to find an appropriate-size free slice. If a slice of the required length is re-used, then its header is also re-used with a new version, and everything works correctly as described above.

However, if we get an allocation request for a larger slice than all available free ones, we might need to merge two (or more) consecutive free slices. Such a merge will cause the slice's data area to "cover" the old header in the second slice . The problem with allowing the application to use (and possibly over-write) the old (de-allocated) header is that old safe-pointers might still refer to it.

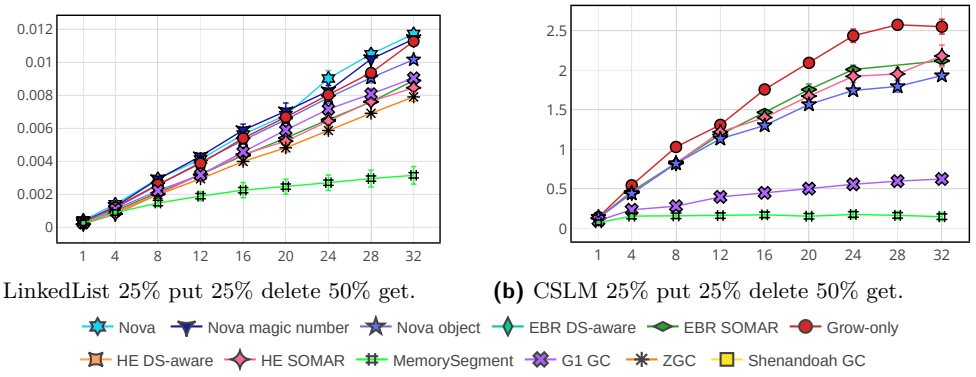
To address this, we add to each header a *magic number*, which is a unique string that is unlikely to occur "in the wild". Specifically, our magic number is a fixed bit string (randomly generated). In our implementation, we use 8 bytes. The magic number is stored in all valid slice headers. When accessing a header, we first verify that the magic number is correct; this is done in addition to validating the version. When we de-allocate a slice, we set its header's magic number field to zero, causing future accesses via this header to fail. The magic number reset can be lazy – we must make it visibly zero only in cases when the slice is merged with a preceding one and the previous header location no longer holds a header.

The magic number approach is probabilistic. It may induce false positives in case (1) a slice is reused and merged with another when an active thread still holds an old reference to it; and (2) the magic number and valid version both occur in the location of a former header by chance. As both fields together comprise $64 + 24 = 88$ bits, the probability for collision is far smaller than the hardware failure rate. This approach is similar to fingerprint-based comparisons used in deduplication techniques [35].

The magic number induces space overhead (it doubles the header size), yet our experiments in the next section show that it has no impact on throughput. The benefit from using magic numbers depends on the workload. If the workload uses uniform allocations and/or entails infrequent deletions, fragmentation is minimal, and the extra overhead for storing magic numbers is not justified. Conversely, if the workload leads to high fragmentation, then the magic number might be beneficial. A study of the fragmentation induced by different workloads is beyond the scope of this paper.

C ARM benchmark results

To check whether there were any platform-specific effects, we repeated some of the experiments on ARM – AWS instance c6gd.8xlarge. The trends were similar to those of x86; results are shown below.



(a) LinkedList 25% put 25% delete 50% get. **(b)** CSLM 25% put 25% delete 50% get.

Figure 7 Throughput (Mops/sec) versus number of threads, on ARM machine.

Improved Deterministic Distributed Maximum Weight Independent Set Approximation in Sparse Graphs

Yuval Gil
Technion – Israel Institute of Technology, Haifa, Israel

Abstract

We design new deterministic CONGEST approximation algorithms for *maximum weight independent set (MWIS)* in *sparse graphs*. As our main results, we obtain new $\Delta(1+\epsilon)$-approximation algorithms as well as algorithms whose approximation ratio depend strictly on α, in graphs with maximum degree Δ and arboricity α. For (deterministic) $\Delta(1+\epsilon)$-approximation, the current state-of-the-art is due to a recent breakthrough by Faour et al. [SODA 2023] that showed an $O(\log^2(\Delta W) \cdot \log(1/\epsilon) + \log^* n)$-round algorithm, where W is the largest node-weight (this bound translates to $O(\log^2 n \cdot \log(1/\epsilon))$ under the common assumption that $W = \text{poly}(n)$). As for α-dependent approximations, a deterministic CONGEST $(8(1+\epsilon)\cdot\alpha)$-approximation algorithm with runtime $O(\log^3 n \cdot \log(1/\epsilon))$ can be derived by combining the aforementioned algorithm of Faour et al. with a method presented by Kawarabayashi et al. [DISC 2020]. As our main results, we show the following.

- A deterministic CONGEST algorithm that computes an $\alpha^{1+\tau}$-approximation for MWIS in $O(\log n \log \alpha)$ rounds for any constant $\tau > 0$. To the best of our knowledge, this is the fastest runtime of any deterministic *non-trivial* approximation algorithm for MWIS to date. Furthermore, for the large class of graphs where $\alpha = \Delta^{1-\Theta(1)}$, it implies a deterministic $\Delta^{1-\Theta(1)}$-approximation algorithm with a runtime of $O(\log n \log \alpha)$ which improves upon the result of Faour et al. in both approximation ratio (by a $\Delta^{\Theta(1)}$ factor) and runtime (by an $O(\log n / \log \alpha)$ factor).
- A deterministic CONGEST algorithm that computes an $O(\alpha)$-approximation for MWIS in $O(\alpha^\tau \log n)$ rounds for any (desirably small) constant $\tau > 0$. This improves the runtime of the best known deterministic $O(\alpha)$-approximation algorithm in the case that $\alpha = O(\text{poly} \log n)$. This also leads to a deterministic $\Delta(1+\epsilon)$-approximation algorithm with a runtime of $O(\alpha^\tau \log n \log(1/\epsilon))$ which improves upon the runtime of Faour et al. in the case that $\alpha = O(\text{poly} \log n)$.
- A deterministic CONGEST algorithm that computes a $(\lfloor(2+\epsilon)\alpha\rfloor)$-approximation for MWIS in $O(\alpha \log n)$ rounds. This improves upon the best known α-dependent approximation ratio by a constant factor.
- A deterministic CONGEST algorithm that computes a $2d^2$-approximation for MWIS in time $O(d^2 + \log^* n)$ in a directed graph with out-degree at most d. The dependency on n is (asymptotically) optimal due to a lower bound by Czygrinow et al. [DISC 2008] and Lenzen and Wattenhofer [DISC 2008].

We note that a key ingredient to all of our algorithms is a novel deterministic method that computes a high-weight subset of nodes whose induced subgraph is sparse.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Approximation algorithms analysis

Keywords and phrases Approximation algorithms, Sparse graphs, The CONGEST model

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.16

Funding This research was supported by VATAT Fund to the Technion Artificial Intelligence Hub (Tech.AI).

Acknowledgements I would like to thank my advisor Yuval Emek for his support. I would also like to thank the anonymous reviewers for their helpful comments.

1 Introduction

The problem of finding a *maximum independent set (MaxIS)* in a graph is long known to be NP-hard. This is because a MaxIS is the complement set of a minimum vertex cover which is among the 21 problems that appear in Karp's seminal work [30].

Further research showed that assuming $NP \neq ZPP$, MaxIS cannot be approximated within $n^{1-\epsilon}$ efficiently for any constant $\epsilon > 0$ [28]. In graphs with maximum degree Δ, a bound of $\Omega(\Delta/\log^2 \Delta)$ on the approximation ratio is shown in [3] assuming the Unique Games Conjecture; whereas a bound of $\Omega(\Delta/\log^4 \Delta)$ is shown in [12] assuming that $P \neq NP$.

On the positive side, efficient approximation algorithms were developed by numerous researchers. An $O(n(\log\log n)^2/\log^3 n)$-approximation is presented in [22]. As for approximations depending on Δ, various studies obtained an $O(\Delta \log\log \Delta/\log \Delta)$-approximation (some of which also apply for the weighted case) [1, 25, 26, 27, 29].

In this paper, we focus on distributed *maximum weight independent set (MWIS)* approximation algorithms. The MWIS problem has been the subject of various studies in both the LOCAL and CONGEST models of distributed computing.

In the LOCAL model, a deterministic $(1+\epsilon)$-approximation algorithm for planar graphs in $O(\log^* n)$ rounds was presented in [16]. For general graphs, the authors of [23] obtained a randomized $(1+\epsilon)$-approximation algorithm in $\operatorname{poly}\log n$ rounds based on network decomposition. This result was later derandomized in [37]. As for lower bounds, it is shown in [16] and [32] that any deterministic $O(\log^* n)$-approximation for (unweighted) MaxIS requires $\Omega(\log^* n)$ rounds. This is extended in [31] to an $\Omega(\log^* n)$ lower bound on the number of rounds required for a randomized algorithm to compute an independent set of size $\Omega(n/\Delta)$.

In the CONGEST model, the main result of [6] is a Δ-approximation in $O(\text{MIS}(G) \cdot \log W)$ rounds, where $\text{MIS}(G)$ is the number of rounds required to find a maximal independent set (MIS) in graph G and W is the largest weight (commonly assumed to be polynomial in n). MWIS was further studied in [31], where a randomized $\Delta(1+\epsilon)$-approximation in $\operatorname{poly}(\log\log n)$ rounds is presented. Deterministic MWIS approximation was considered recently in [21]. Among other results, a deterministic $B(1+\epsilon)$-approximation in $O(\log^2(\Delta W)\log(\epsilon^{-1}) + \log^* n)$ rounds is presented, where B is the neighborhood independence. On the negative side, hardness results for exact and approximate MaxIS computation in the CONGEST model are shown in [11, 4, 19].

1.1 Our Objective and Motivation

In this paper, our goal is to study deterministic CONGEST algorithms for the MWIS problem in *sparse graphs*. Distributed graph algorithms in sparse graphs have been a focal point of much research since they were considered in the seminal work of Goldberg et al. [24]. In [9], Barenboim and Elkin presented fast deterministic algorithms for some classical symmetry-breaking problems on *bounded arboricity* graphs. The arboricity of a graph, denoted by α, is the minimum number of edge disjoint forests into which the edges of the graph can be partitioned. The rich class of bounded arboricity graphs include many well-studied and important graph families such as planar graphs, graphs of bounded treewidth, and graphs excluding a fixed minor.

In recent years, there is a growing interest in the class of bounded arboricity graphs (and its subclasses) in the context of *distributed optimization*. As it turns out, it is often the case that bounded arboricity graphs allow for a better approximation and/or faster algorithms than in general graphs. Hence, a plethora of research has been devoted to distributed

algorithms that operate on bounded arboricity graphs for various classical optimization problems such as maximum matching, minimum dominating set, and minimum vertex cover (see, e.g., [13, 18, 14, 15, 16, 17, 33, 35, 2, 20]).

The motivation for studying MWIS in bounded arboricity graphs is similar – in the common case, the approximation ratio (and natural barrier) obtained by CONGEST algorithms is linear in Δ (which bounds α from above). Therefore, approximation algorithms that depend strictly on α are favorable for a large class of graphs. In light of that, Kawarabayashi et al. [31] showed that given a CONGEST $\Delta(1+\epsilon)$-approximation algorithm $\mathcal{A}$ with runtime $T(n,\Delta)$ (for n-node graph with maximum degree Δ), there is a CONGEST $8(1+\epsilon)\alpha$-approximation algorithm for MWIS in time $O(T(n,O(\alpha))\cdot\log n)$.[1] By the work of [31], this leads to an $O(\log n\cdot\text{poly}(\log\log n)/\epsilon)$-round randomized algorithm. Regarding deterministic algorithms, plugging in the aforementioned result of [21] implies an $O(\log^3 n\cdot\log(\epsilon^{-1}))$-round algorithm; whereas plugging in the deterministic $O(\Delta+\log^* n)$-round algorithm of [6] implies an $O(\log n\cdot(\alpha+\log^* n))$-round algorithm.

1.2 Our Contributions

In this paper, we present new distributed deterministic approximation algorithms for the *maximum weight independent set (MWIS)* problem. All of our algorithms operate in the CONGEST model (see Section 2 for a definition). A key ingredient in our algorithms is a new procedure called `Sparse_Set` which is outlined below.

Outline of Procedure `Sparse_Set`. The goal of the `Sparse_Set` procedure is simple: given a graph $G=(V,E)$ with node-weight function $w:V\to\mathbb{R}_{\geq 0}$, we seek to compute a subset $X\subseteq V$ with large weight $w(X)=\sum_{v\in X}w(v)$ whose induced subgraph $G(X)$ is relatively sparse. To achieve this goal, `Sparse_Set` relies on a new notion called *β-bounded coloring*. A β-bounded coloring is a proper node-coloring such that each node has at most β neighbors with larger colors for some integer parameter $\beta>0$ (refer to Section 2 for a formal definition). The `Sparse_Set` procedure takes as input a β-bounded coloring c of graph G and an integer parameter $1\leq f\leq\beta$, and returns a subset $X\subseteq V$. The properties of `Sparse_Set` are specified in the following lemma.

▶ **Lemma 1.1.** *Let $G=(V,E)$ be a graph with node-weight function $w:V\to\mathbb{R}_{\geq 0}$, let c be a β-bounded coloring of G and let $1\leq f\leq\beta$ be an integer parameter. Upon termination,* `Sparse_Set`(c,f) *returns a subset $X\subseteq V$ that satisfies: (1) $|X\cap L(v)|<\beta/f$ for each selected node $v\in X$, where $L(v)=\{u\in N(v)\mid c(u)>c(v)\}$ is the set of v's neighbors with larger color; (2) $f\cdot w(X)\geq OPT(G)$, where $OPT(G)$ is the weight of a MWIS in G; and (3) $2\cdot f\cdot w(X)\geq w(V)$. The runtime of* `Sparse_Set`$(c,f)$ *is $O(k)$, where k is the total number of distinct colors assigned by the coloring c.*

Section 3 is mostly dedicated to proving Lemma 1.1. This is done by means of a *primal-dual* approach.

Our bounds. In Sections 4 and 5, we use `Sparse_Set` in various ways to construct new approximation algorithms. Refer to Table 1 for a list of our bounds.

In Section 4, we consider α-arboricity graphs. First, using `Sparse_Set` in a somewhat straightforward manner, we get the following theorem.

[1] We believe that the approximation ratio stated in [31] can be improved to $4(1+\epsilon)$ without affecting the (asymptotic) runtime by a slightly different choice of constants.

Table 1 A list of our MWIS approximations. Here, α denotes the arboricity and Δ denotes the maximum degree. Runtime improvements for $O(\alpha)$-approximation algorithm are compared with the deterministic $O(\log^3 n)$-round algorithm (assuming $W = \text{poly}(n)$) derived from [31] and [21] (see Section 1.1 for more details).

Approx.	Runtime	Notes
$\alpha^{1+\tau}$	$O(\log n \log \alpha)$	for any constant $\tau > 0$
$\Delta^{1-\Theta(1)}$	$O(\log n \log \alpha)$	restricted to graphs where $\alpha = \Delta^{1-\Theta(1)}$; improves approx. ratio of [21] by a $\Delta^{\Theta(1)}$ factor; improves runtime of [21] by an $O(\log n / \log \alpha)$ factor
$O(\alpha)$	$O(\alpha^\tau \log n)$	for any constant $\tau > 0$; improves runtime of [31, 21] when $\alpha = O(\text{poly} \log n)$
$\Delta(1+\epsilon)$	$O(\alpha^\tau \log n \log(1/\epsilon))$	for any constant $\tau > 0$; improves runtime of [21] when $\alpha = O(\text{poly} \log n)$
$\lfloor (2+\epsilon) \cdot \alpha \rfloor$	$O(\alpha \log n)$	improves approx. ratio of [31]
$O(\alpha^2)$	$O(\log n + \sqrt{\alpha \log n}$ $+\alpha^{3/4} \log \alpha)$	
$2d^2$	$O(d^2 + \log^* n)$	directed graphs with out-degree $\leq d$; $O(\log^* n)$ is necessary due to [16, 32]

▶ **Theorem 1.2.** *For any constant $\epsilon > 0$, there exists a deterministic CONGEST algorithm that computes a $(\lfloor (2+\epsilon) \cdot \alpha \rfloor)$-approximation for MWIS in $O(\alpha \log n)$ rounds.*

By a simple modification, we also establish the following theorem.

▶ **Theorem 1.3.** *There exists a deterministic CONGEST algorithm that computes an $O(\alpha^2)$-approximation for MWIS in $O(\log n + \texttt{COL}(n, (2+\epsilon)\cdot\alpha) + \sqrt{\alpha \log n})$ rounds, where $\texttt{COL}(n, \Delta)$ is the runtime of $(\Delta+1)$-coloring an n-node graph with maximum degree Δ.*

For example, plugging in the $(\Delta+1)$-coloring algorithm of [7] leads to a runtime bound of $O(\log n + \alpha^{3/4} \log \alpha + \sqrt{\alpha \log n})$.

We then relax the $\lfloor (2+\epsilon) \cdot \alpha \rfloor$ approximation ratio of Theorem 1.2 in favor of faster algorithms. Using `Sparse_Set` in a slightly more elaborate way and pairing it with an arbdefective coloring (refer to Section 2 for a definition) algorithm of [10], we obtain the following lemma.

▶ **Lemma 1.4.** *For any integer $k > 0$, there exists a deterministic CONGEST algorithm that computes an $(8^k \cdot \alpha)$-approximation for MWIS in $O(k \cdot \alpha^{1/k} \cdot \log n)$ rounds.*

As a consequence of Lemma 1.4, we obtain the following four theorems.

▶ **Theorem 1.5.** *For any constant $\tau > 0$, there exists a deterministic CONGEST algorithm that computes an $O(\alpha)$-approximation for MWIS in $O(\alpha^\tau \log n)$ rounds.*

▶ **Theorem 1.6.** *Let $\epsilon > 0$ be a parameter. For any constant $\tau > 0$, there exists a deterministic CONGEST algorithm that computes a $\Delta(1+\epsilon)$-approximation for MWIS in $O(\alpha^\tau \log n \log(1/\epsilon))$ rounds.*

▶ **Theorem 1.7.** *For any constant $\tau > 0$, there exists a deterministic CONGEST algorithm that computes an $\alpha^{1+\tau}$-approximation for MWIS in $O(\log \alpha \log n)$ rounds.*

▶ **Theorem 1.8.** *For any graph $G = (V, E)$ with arboricity α and maximum degree Δ such that $\alpha = \Delta^{1-\Theta(1)}$, there exists a deterministic CONGEST algorithm that computes a $\Delta^{1-\Theta(1)}$-approximation for MWIS in $O(\log \alpha \log n)$ rounds.*

We note that the time complexity stated in Theorems 1.5 and 1.6 improve the state-of-the-art for approximations of $O(\alpha)$ and $\Delta(1+\epsilon)$, respectively, on graphs with $\alpha = O(\text{poly}\log n)$. Additionally, to the best of our knowledge, the $O(\log\alpha \log n)$ runtime of Theorems 1.7 and 1.8 is the fastest known runtime of a deterministic non-trivial approximation algorithm for MWIS.

Finally, in Section 5, we consider directed graphs and obtain the following theorem.

▶ **Theorem 1.9.** *For a directed graph $G = (V, E)$ with out-degree at most d, there exists a deterministic CONGEST algorithm that computes a $2d^2$-approximation for MWIS in $O(d^2 + \log^* n)$ rounds.*

We remark that the $O(\log^* n)$ term in the time complexity is asymptotically tight due to an existing lower bound presented in [16, 32] (refer to Section 5 for more details).

2 Preliminaries

Consider a graph $G = (V, E)$ and denote $n = |V|$ and $m = |E|$. For each node $v \in V$, we denote by $N(v)$ the set of v's *neighbors* in G. Let $\deg(v) = |N(v)|$ be the *degree* of node v and let us denote by $\Delta = \max_{v\in V}\{\deg(v)\}$ the largest degree in the graph. If G is directed, then we use the notation $(u \to v)$ to reflect that the edge $(u, v) \in E$ is directed from u to v. In this context, the notation (u, v) (or (v, u)) refers to an edge between u and v that could be directed in either direction. For a node $v \in V$, we say that a node $u \in N(v)$ is an *incoming* (resp., *outgoing*) neighbor of v if there exists an edge $(u \to v) \in E$ (resp., $(v \to u) \in E$)). The *in-degree* (resp., *out-degree*) of node $v \in V$ is defined to be the number of v's incoming (resp., outgoing) neighbors. For a node subset $U \subseteq V$, let $G(U)$ denote the subgraph induced by U.

The CONGEST model. Our algorithms operate in the CONGEST model [36], where a communication network is abstracted by an n-node graph $G = (V, E)$. Each node $v \in V$ is equipped with a unique $O(\log n)$-bit identifier. Computation progresses in synchronous communication rounds, where each node $v \in V$ may send a message of size $O(\log n)$ to each neighbor $u \in N(v)$. If G is directed, then the direction of each edge $(u, v) \in E$ is encoded to the endpoints u and v by means of a consistent orientation function. Notice that communication may occur on both directions of the edge (u, v) regardless of its orientation.

Maximum weight independent set. Consider a graph $G = (V, E)$. A subset $X \subseteq V$ of nodes is said to be an *independent set* of G if it holds that $(u, v) \notin E$ for all $u, v \in X$. For a node-weight function $w : V \to \mathbb{R}_{\geq 0}$, a *maximum weight independent set (MWIS)* is an independent set $X \subseteq V$ that maximizes $w(X) := \sum_{v\in X} w(v)$.[2] We denote by $OPT(G)$ the weight of a MWIS in G. As usual, for a parameter $q \geq 1$, we say that a subset $X \subseteq V$ of nodes is a *q-approximation* for MWIS if it is an independent set that satisfies $q \cdot w(X) \geq OPT(G)$.

In a natural linear program (LP) relaxation of MWIS, each node $v \in V$ is associated with a variable x_v. The objective is to maximize $\sum_{v\in V} w(v) \cdot x_v$, subject to the constraints $x_u + x_v \leq 1$ for each edge $(u, v) \in E$; and $x_v \geq 0$ for each node $v \in V$. In the dual LP, each edge $e \in E$ is associated with a variable y_e. The dual objective is to minimize $\sum_{e\in E} y_e$, subject to the constraints $\sum_{u\in N(v)} y_{u,v} \geq w(v)$ for each node $v \in V$; and $y_e \geq 0$

[2] Throughout this paper, we stick to the common assumption that all assigned weights can be represented using $O(\log n)$ bits and thus can be sent by means of a single message in the CONGEST model.

for each edge $e \in E$. The *weak duality* theorem [38, Chapter 12, Theorem 12.2] implies that $w(X) \leq \sum_{e \in E} y_e$ for any independent set $X \subseteq V$ and feasible dual solution $\mathbf{y} = \{y_e\}_{e \in E}$ (i.e., a dual solution that satisfies all constraints).

β-Bounded coloring. Given a graph $G = (V, E)$ and integers $\beta, k > 0$, we say that a node-coloring $c : V \to [k]$ is *β-bounded* if the following conditions are satisfied: (1) c is a *proper* coloring, i.e., $c(u) \neq c(v)$ for all $(u, v) \in E$; and (2) each node $v \in V$, has at most β neighbors with larger color, i.e., the set $L(v) = \{u \in N(v) \mid c(u) > c(v)\}$ satisfies $|L(v)| \leq \beta$ for all $v \in V$.[3]

Arboricity. Given an undirected graph $G = (V, E)$, the *arboricity* of G is defined to be the smallest integer $\alpha > 0$ for which there exists a partition $E_1, \ldots, E_\alpha$ of the edges into α pairwise-disjoint sets such that (V, E_i) is a forest for each $i \in [\alpha]$.

Barenboim and Elkin's Partition Procedure. In Section 4 we make use of a partition procedure presented by Barenboim and Elkin in [9]. This procedure takes as parameters the graph's arboricity α and a constant $\epsilon > 0$. We shall refer to this procedure as `BE_Partition`(α, ϵ). Procedure `BE_Partition`(α, ϵ) partitions the nodes of graph G into $\ell = O(\log n)$ layers $V = H_1 \dot\cup \ldots \dot\cup H_\ell$. For all $i \in [\ell]$, it is guaranteed that each node $v \in H_i$ has at most $\lfloor (2 + \epsilon) \cdot \alpha \rfloor$ neighbors in $\cup_{j=i}^{\ell} H_j$. As established in [9], `BE_Partition` takes $O(\log n)$ rounds in the CONGEST model.

Arbdefective coloring. The notion of *d-arbdefective* coloring was introduced by Barenboim and Elkin in [8]. We say that a coloring $c : V \to [k]$ of graph $G = (V, E)$ is d-arbdefective if for every color $i \in [k]$, the subgraph $G(V_i)$ induced by the subset $V_i = \{v \in V \mid c(v) = i\}$, has arboricity at most d. In [10], it is shown that for any $1 \leq p \leq \Delta$, a (Δ/p)-arbdefective coloring that uses $O(p)$ colors can be computed in $O(p + \log^* n)$ rounds in the CONGEST model.

3 The `Sparse_Set` Procedure

In this section, we present a simple procedure referred to as `Sparse_Set` (Algorithm 1). Let $G = (V, E)$ be a graph with node-weight function $w : V \to \mathbb{R}_{\geq 0}$ and let $\beta \in \mathbb{Z}_{>0}$. The procedure takes as input a graph G, along with a β-bounded k-coloring c, and an integer parameter f (encoded to each node) such that $1 \leq f \leq \beta$, and returns a subset $X \subseteq V$ of *selected* nodes. The properties of the set X as well as the runtime of procedure `Sparse_Set` are captured by the following lemma.

▶ **Lemma 3.1.** *Upon termination,* `Sparse_Set`(c, f) *returns a subset $X \subseteq V$ that satisfies: (1) $|X \cap L(v)| < \beta/f$ for each selected node $v \in X$, where $L(v) = \{u \in N(v) \mid c(u) > c(v)\}$ is the set of v's neighbors with larger color; (2) $f \cdot w(X) \geq OPT(G)$; and (3) $2 \cdot f \cdot w(X) \geq w(V)$. The runtime of* `Sparse_Set`$(c, f)$ *is $O(k)$, where k is the total number of distinct colors assigned by the coloring c.*

We now describe the procedure `Sparse_Set`. Refer to Algorithm 1 for a pseudocode description.

[3] We sometimes naturally extend this definition to a coloring that assigns colors chosen from some ordered set of k elements (and not necessarily $[k]$).

Algorithm 1 Procedure `Sparse_Set`(c, f) from the perspective of node $v \in V$.

Input: integer $1 \leq f \leq \beta$;
colors $c(v)$ and $c(u)$ for each $u \in N(v)$ (c is a β-bounded coloring)

1: $v.y_{u,v} = \perp$ for all $u \in N(v)$ ▷ v's dual variables
2: $L(v) = \{u \in N(v) \mid c(u) > c(v)\}$ ▷ v's neighbors with larger color
3: $S(v) = N(v) - L(v)$ ▷ v's neighbors with smaller color
4: $IN(v) = OUT(v) = \emptyset$
5: $\lambda(v) = \perp$
6: $v.status = undecided$
7: **while** $\lambda(v) == \perp$ **do** ▷ first stage
8: **for** each $u.y_{u,v}$ received from a neighbor $u \in S(v)$ **do**
9: $v.y_{u,v} = u.y_{u,v}$
10: **if** $v.y_{u,v} \neq \perp$ for all $u \in S(v)$ **then**
11: $\lambda(v) = \max\{0, w(v) - \sum_{u \in S(v)} v.y_{u,v}\}$
12: $v.y_{u,v} = \frac{\lambda(v) \cdot f}{|L(v)|}$ for all $u \in L(v)$
13: send $v.y_{u,v}$ to all $u \in L(v)$
14: **if** $\lambda(v) == 0$ **then**
15: $v.status = eliminated$
16: send 'eliminated' to all $u \in S(v)$
17: **while** $v.status == undecided$ **do** ▷ second stage
18: **for** each 'eliminated' received from a neighbor $u \in L(v)$ **do**
19: $OUT(v) = OUT(v) \cup \{u\}$
20: **for** each 'selected' received from a neighbor $u \in L(v)$ **do**
21: $IN(v) = IN(v) \cup \{u\}$
22: **if** $IN(v) \cup OUT(v) == L(v)$ **then** ▷ if all nodes in $L(v)$ are decided
23: **if** $|IN(v)| \geq \frac{|L(v)|}{f}$ **then**
24: $v.status = eliminated$
25: send 'eliminated' to all $u \in S(v)$
26: **else**
27: $v.status = selected$
28: send 'selected' to all $u \in S(v)$

Overview of Algorithm 1. Throughout the execution of procedure `Sparse_Set`, each node $v \in V$ maintains a status $v.status \in \{undecided, selected, eliminated\}$ and a dual variable $y_{u,v}$ for each neighbor $u \in N(v)$. We emphasize that the dual variables are used for analysis purpose. In particular, upon termination the dual solution $\mathbf{y}$ computed during procedure `Sparse_Set` might be infeasible (we address that later in the analysis). Initially, each node sets its status to *undecided* and all dual variables to $\perp$. For each node $v \in V$, define $L(v) = \{u \in N(v) \mid c(u) > c(v)\}$ and $S(v) = N(v) - L(v)$ to be the sets of v's neighbors with larger and smaller colors, respectively. Additionally, each node v maintains a numerical value $\lambda(v)$ initialized to $\perp$.

The execution of the algorithm can be divided into two subsequent stages: the first stage is dedicated to computing the dual variables (although some nodes may also be eliminated at this stage); whereas the goal of the second stage is for each undecided node to decide whether it becomes eliminated or selected.

During the first stage (lines 7–16), each node $v \in V$ waits until it receives a dual variable $y_{u,v}$ from every neighbor $u \in S(v)$ (notice that if $S(v) = \emptyset$, then v does not need to wait). Upon receiving all dual variables, v sets $\lambda(v) = \max\{0, w(v) - \sum_{u \in S(v)} y_{u,v}\}$. Then, for each $u \in L(v)$, v sets $y_{u,v} = \frac{\lambda(v) \cdot f}{|L(v)|}$ and informs u. If $\lambda(v) = 0$, then v becomes eliminated and informs all of its neighbors $u \in S(v)$.

In the second stage (lines 17–28), all nodes become eliminated or selected. Each undecided node v waits until it receives a message $\mu_u \in \{\text{'eliminated'}, \text{'selected'}\}$ from every neighbor $u \in L(v)$. After receiving a message from every $u \in L(v)$, each undecided node v becomes eliminated if at least $|L(v)|/f$ of its neighbors in $L(v)$ became selected; otherwise, v becomes selected (notice that if $L(v) = \emptyset$, then v becomes selected in the second stage if and only if it was not eliminated in the first stage). After changing its status, v informs its neighbors $u \in S(v)$ of the new status.

We now analyze the procedure `Sparse_Set`. Let us denote by $X = \{v \in V \mid v.status = selected\}$ the set of nodes that were selected during the algorithm and by $\mathbf{y} = \{y_{u,v} \mid (u,v) \in E\}$ the dual solution.

Observe that the correctness of Property (1) of Lemma 3.1 follows directly from the construction of `Sparse_Set` and the fact that $|L(v)| \leq \beta$. Hence, our goal now is to prove properties (2) and (3). To that end, for the sake of analysis, let us define a graph $G' = (V', E')$ as the graph obtained from G by adding a virtual zero-weighted node z and connecting it by edges to all nodes $v \in V$ with $L(v) = \emptyset$. That is, G' consists of node set $V' = V \cup \{z\}$, edge set $E' = E \cup \{(z,v) \mid L(v) = \emptyset\}$, and weights $w'(z) = 0$ and $w'(v) = w(v)$ for all $v \in V$. We also extend the definitions of $\mathbf{y}$ and the sets $L(v)$ to the graph G' as follows. For each node $v \in V$ with $L(v) = \emptyset$ in G, we define $L'(v) = \{z\}$ and $y'_{v,z} = \lambda(v) \cdot f$; for nodes $v \in V$ with $L(v) \neq \emptyset$ in G, we keep $L'(v) = L(v)$ and $y'_{u,v} = y_{u,v}$ for each $u \in N(v)$. Let $N'(v)$ denote the set of v's neighbors in G' for each node $v \in V'$.

▶ **Observation 3.2.** *$OPT(G) = OPT(G')$.*

Proof. Clearly, $OPT(G) \leq OPT(G')$ as any independent set of G is also an independent set of G'. As for the other direction, consider a MWIS $I \subseteq V'$ of G'. The set $I \cap V$ is an independent set of G with weight $w(I \cap V) = w(I)$. Thus, we get that $OPT(G) \geq w(I \cap V) = w(I) = OPT(G')$. ◀

We note that Observation 3.2 implies that to establish Property (2) of Lemma 3.1, it is sufficient to bound $w(X)$ in terms of $OPT(G')$. Furthermore, by weak duality, it is sufficient to bound $w(X)$ in terms of a feasible dual solution for G'. To that end, we show the following.

▶ **Observation 3.3.** *$\mathbf{y}'$ is a feasible dual solution for G'.*

Proof. First consider the virtual node z. Since $\mathbf{y}'$ is non-negative, it follows that

$$\sum_{u \in N'(z)} y_{u,z} \geq w'(z) = 0 \ .$$

As for nodes $v \neq z$, we note that by construction, it follows that

$$\sum_{u \in L'(v)} y_{u,v} = \sum_{u \in L'(v)} \frac{\lambda(v) \cdot f}{|L'(v)|} = \lambda(v) \cdot f \geq \lambda(v) \ .$$

Feasibility follows since $\lambda(v) = \max\{0, w(v) - \sum_{u \in S(v)} v.y_{u,v}\} \geq w(v) - \sum_{u \in S(v)} v.y_{u,v}$, which implies $\sum_{u \in N'(v)} y'_{u,v} = \sum_{u \in S(v)} y'_{u,v} + \sum_{u \in L'(v)} y'_{u,v} \geq w(v)$ for each node $v \in V$. ◀

We partition the nodes of $V - X$ into two sets $R_1 = \{v \in V - X \mid \sum_{u \in S(v)} y'_{u,v} \geq w(v)\}$ and $R_2 = V - X - R_1$. Observe that R_1 and R_2 are the sets of nodes that were eliminated in the first and second stage, respectively.

The following simple observation follows directly from the construction of $\mathbf{y}'$.

▶ **Observation 3.4.** *Consider a node $v \in R_1$. For each $u \in L'(v)$, it holds that $y'_{u,v} = 0$.*

From Observation 3.4, we can derive the following equality regarding the dual objective value.

▶ **Corollary 3.5.** $\sum_{e\in E'} y'_e = \sum_{v\in X\cup R_2} \sum_{u\in L'(v)} y'_{u,v}$

Proof. First, notice that $\sum_{e\in E'} y'_e = \sum_{v\in V} \sum_{u\in L'(v)} y'_{u,v}$. The assertion follows since

$$\sum_{v\in V} \sum_{u\in L'(v)} y'_{u,v} = \sum_{v\in X\cup R_2} \sum_{u\in L'(v)} y'_{u,v} + \sum_{v\in R_1} \sum_{u\in L'(v)} y'_{u,v} = \sum_{v\in X\cup R_2} \sum_{u\in L'(v)} y'_{u,v}$$ ◀

We make the following observation regarding R_2.

▶ **Observation 3.6.** *For each node $v \in R_2$, it holds that $|X \cap L'(v)| = |X \cap L(v)| \geq \frac{|L'(v)|}{f}$.*

Proof. First, observe that nodes $v \in V$ with $L'(v) = \{z\} \neq L(v)$ are never eliminated in the second stage (they are either eliminated in the first stage or selected in the second stage). Hence, every node $v \in R_2$ satisfies $L(v) = L'(v)$ and thus $|X \cap L'(v)| = |X \cap L(v)|$. Now, for each node $v \in R_2$, the inequality $|X \cap L(v)| \geq \frac{|L'(v)|}{f}$ follows immediately from the fact that v was eliminated in the second stage. ◀

We are now ready to bound $w(X)$ in terms of the objective value of $\mathbf{y}'$.

▶ **Lemma 3.7.** $f \cdot w(X) \geq \sum_{e\in E} y'_e$.

Proof. For each node $v \in R_2$, (arbitrarily) define a partition of $L'(v)$ into $|X \cap L'(v)|$ disjoint sets of at most f nodes each. Observe that such partition exists since $f \cdot |X \cap L'(v)| \geq |L'(v)|$ by Observation 3.6. We now identify each such set with a node $u \in X \cap L'(v)$, and denote this set by $\mu_v(u)$. Now, for every $v \in X$, let $\rho(v) = \sum_{u\in S(v)\cap R_2} \sum_{u'\in\mu_u(v)} y'_{u,u'}$ and observe that by the construction of $\mathbf{y}'$, it holds that $\rho(v) = \sum_{u\in S(v)\cap R_2} |\mu_u(v)| \cdot \frac{\lambda(u)\cdot f}{|L'(u)|} \leq \sum_{u\in S(v)\cap R_2} f \cdot y'_{u,v} = f \cdot \sum_{u\in S(v)\cap R_2} y'_{u,v}$.

Recall that Corollary 3.5 states that $\sum_{e\in E'} y'_e = \sum_{v\in X\cup R_2} \sum_{u\in L'(v)} y'_{u,v}$. Developing further, we get

$$\begin{aligned}
\sum_{v\in X\cup R_2} \sum_{u\in L'(v)} y'_{u,v} &= \sum_{v\in X} \sum_{u\in L'(v)} y'_{u,v} + \sum_{v\in R_2} \sum_{u\in L'(v)} y'_{u,v} = \\
&\sum_{v\in X} \sum_{u\in L'(v)} \frac{\lambda(v)\cdot f}{|L'(v)|} + \sum_{v\in R_2} \sum_{u\in X\cap L'(v)} \sum_{u'\in\mu_v(u)} y'_{u',v} = \\
&\sum_{v\in X} \lambda(v)\cdot f + \sum_{v\in X} \sum_{u\in S(v)\cap R_2} \sum_{u'\in\mu_u(v)} y'_{u',u} = \\
&\sum_{v\in X} \lambda(v)\cdot f + \sum_{v\in X} \rho(v) \leq \\
&f \cdot \sum_{v\in X} \left(\lambda(v) + \sum_{u\in S(v)\cap R_2} y'_{u,v}\right) \leq \\
f \cdot \sum_{v\in X} \left(w(v) - \sum_{u\in S(v)\cap R_2} y'_{u,v} + \sum_{u\in S(v)\cap R_2} y'_{u,v}\right) &= f \cdot w(X),
\end{aligned}$$

where the second sum in the second line holds because by definition, $L'(v) = \bigcup_{u\in X\cap L'(v)} \mu_v(u)$ for each $v \in R_2$, and the penultimate inequality holds because by construction, $\lambda(v) = w(v) - \sum_{u\in S(v)} y'_{u,v} \leq w(v) - \sum_{u\in S(v)\cap R_2} y'_{u,v}$ for each $v \in X$. ◀

It is now simple to show Property (3).

▶ **Lemma 3.8** (Property 3). $2 \cdot f \cdot w(X) \geq w(V)$.

Proof. It follows from Lemma 3.7 that $f \cdot w(X) \geq \sum_{e \in E} y'_e$. Thus, it suffices to show that $2 \cdot \sum_{e \in E} y'_e \geq w(V)$. Since $\mathbf{y}'$ is feasible, it holds that $\sum_{u \in N'(v)} y'_{u,v} \geq w(v)$ for each $v \in V'$. Therefore, we get $2 \cdot \sum_{e \in E} y'_e = \sum_{v \in V'} \sum_{u \in N'(v)} y'_{u,v} \geq \sum_{v \in V'} w(v) = w(V') = w(V)$, where the first equality holds because in the summation $\sum_{v \in V'} \sum_{u \in N'(v)} y'_{u,v}$, we go over every edge twice. ◀

We are now prepared to prove Lemma 3.1.

Proof of Lemma 3.1. As noted before, Property (1) follows directly from the `Sparse_Set` construction. Property (2) follows from Observations 3.2 and 3.3 and Lemma 3.7. Property (3) is established in Lemma 3.8.

Regarding runtime, let us bound the number of rounds in each stage. By a simple inductive argument, it holds that after at most i rounds of the first stage, each node v colored by the i-th smallest color receives a dual variable $y_{u,v}$ from each neighbors $u \in S(v)$. Thus, the first stage finishes after at most k rounds. Similarly, after at most i rounds of the second stage, each node v colored by the i-th largest color receives a message $\mu_u \in \{\text{'eliminated'}, \text{'selected'}\}$ from every neighbor $u \in L(v)$. Hence, the second stage also finishes after at most k rounds. Overall, we get that the total runtime is $O(k)$. ◀

We make another simple observation regarding the induced subgraph $G(X)$ that would be useful for the results obtained in later sections.

▶ **Observation 3.9.** *The arboricity of $G(X)$ is smaller than β/f.*

Proof. Let ρ be the orientation obtained by directing each edge of $G(X)$ towards the endpoint that has a larger color according to c. Due to Property (1) of Lemma 3.1, we get that every node $v \in X$ has less than β/f outgoing edges in $G(X)$ under the orientation ρ. That is, the edges of $G(X)$ can be partitioned into $q < \beta/f$ subsets $E_1, \ldots, E_q$ of edges such that for each subset E_i, the out-degree of each node $v \in X$ is at most 1. Moreover, by construction, each subgraph (X, E_i) is acyclic with respect to ρ. Thus, each subgraph (X, E_i) is a forest. Overall, we get that the edges of $G(X)$ can be partitioned into q forests. ◀

Notice that it follows from Lemma 3.1 that in the case that `Sparse_Set` is invoked with $f = \beta$, the algorithm returns an independent set X which is a β-approximation for MWIS.[4] That is, the following lemma is a special case of Lemma 3.1.

▶ **Lemma 3.10.** *Given a β-bounded coloring c of graph $G = (V, E)$, there exists an algorithm that computes a β-approximation $X \subseteq V$ for MWIS. Moreover, it holds that $2 \cdot \beta \cdot w(X) \geq w(V)$. The runtime of this algorithm is $O(k)$, where k is the total number of distinct colors assigned by c.*

▶ Remark 3.11. In fact, to obtain the β-approximation of Lemma 3.10, it suffices for each node $v \in V$ to replace f with $|L(v)|$ (instead of with β). This modification does not affect the correctness nor the runtime. Hence, Lemma 3.10 can be accomplished even without the nodes knowing β.

[4] We note that this special case of the `Sparse_Set` procedure can be derived from the local-ratio approach presented in [5].

By some simple modifications, we are also able to devise a faster approximation algorithm (Algorithm 2) while incurring a quadratic increase to the approximation guarantee of Lemma 3.10. Specifically, we establish the following lemma.

▶ **Lemma 3.12.** *Given a β-bounded coloring $c : V \to [k]$, Algorithm 2 computes a $2\beta^2$-approximation for MWIS. The runtime of Algorithm 2 is $O(\sqrt{k})$.*

The idea of Algorithm 2 is very simple. We divide the color $c(v) \in [k]$ of each node $v \in V$ to two colors $c_1(v), c_2(v) \in \{0, 1, \ldots, \lfloor\sqrt{k}\rfloor\}$ such that $c(v) = \lceil\sqrt{k}\rceil \cdot c_1(v) + c_2(v)$. Then, the algorithm computes a set $X \subseteq V$ by invoking the algorithm derived from Lemma 3.10 with the coloring c_1 on the graph $G_1 = (V, E_1)$, where $E_1 = \{(u, v) \mid c_1(u) \neq c_1(v)\}$. Finally, to compute the output set X', the algorithm of Lemma 3.10 is invoked again, this time with the coloring c_2 on the induced subgraph $G(X)$.

Algorithm 2 A $2\beta^2$-approximation algorithm for MWIS on a graph $G = (V, E)$ with given β-bounded coloring $c : V \to [k]$.

1: let $(c_1(v), c_2(v)) \in \{0, 1, \ldots, \lfloor\sqrt{k}\rfloor\}^2$ s.t. $c(v) = \lceil\sqrt{k}\rceil \cdot c_1(v) + c_2(v)$ for each $v \in V$
2: let $E_1 = \{(u, v) \mid c_1(u) \neq c_1(v)\}$ ▷ bi-chromatic edges according to c_1
3: run `Sparse_Set`(c_1, β) on $G_1 = (V, E_1)$ to obtain set $X \subseteq V$
4: run `Sparse_Set`(c_2, β) on $G(X)$ to obtain set X'
5: return X' as a MWIS approximation

We now prove Lemma 3.12.

Proof of Lemma 3.12. Regarding runtime, we invoke `Sparse_Set` with c_1 and c_2. Since both coloring functions use $O(\sqrt{k})$ colors, the runtime complexity is $O(\sqrt{k})$.

As for correctness, we start by showing that c_1 is a β-bounded coloring on G_1. Notice that by definition, c_1 is proper with respect to G_1. To see that it is β-bounded, observe that $c_1(u) > c_1(v) \implies c(u) > c(v)$. Since c is β-bounded, so is c_1. We can similarly show that c_2 is a β-bounded coloring with respect to $G(X)$. First, note that the edges of $G(X)$ must be monochromatic with respect to c_1. Since c is a proper coloring, the edges of $G(X)$ must be bi-chromatic with respect to c_2. Moreover, it follows that $c_2(u) > c_2(v) \implies c(u) > c(v)$ which implies that c_2 is β-bounded in $G(X)$. The approximation ratio now follows since $2\beta \cdot w(X') \geq w(X) \geq OPT(G)/\beta \implies 2\beta^2 \cdot w(X') \geq OPT(G)$. ◀

4 Approximation Algorithms for α-Arboricity Graphs

In this section we present new deterministic approximation algorithms for MWIS on graphs $G = (V, E)$ with arboricity α. For ease of presentation, we present our algorithms under the assumption that every node $v \in V$ knows the value of α. Refer to Appendix A for a discussion on how this assumption can be lifted at the cost of a multiplicative factor of at most $O(\log \alpha)$ to the runtime.

4.1 A Basic $(\lfloor(2+\epsilon)\cdot\alpha\rfloor)$-Approximation Algorithm

In this section, we present an algorithm that computes a $(\lfloor(2+\epsilon)\cdot\alpha\rfloor)$-approximation for MWIS. More concretely, we prove the following theorem.

▶ **Theorem 4.1.** *For any constant $\epsilon > 0$, Algorithm 3 computes a $(\lfloor(2+\epsilon)\cdot\alpha\rfloor)$-approximation $X \subseteq V$ for MWIS. Moreover, $2 \cdot (\lfloor(2+\epsilon)\cdot\alpha\rfloor) \cdot w(X) \geq w(V)$. The runtime of Algorithm 3 is $O(\alpha \log n)$.*

We now describe the algorithm. Moving forward, we shall use the notation $\delta = \lfloor (2+\epsilon)\cdot\alpha \rfloor$. Refer to Algorithm 3 for a pseudocode description.

Algorithm 3 A δ-approximation algorithm for an α-arboricity graph $G = (V, E)$ ($\delta = \lfloor (2+\epsilon)\cdot\alpha \rfloor$).

1: run `BE_Partition`(α, ϵ) ▷ computes node-partition into layers $H_1, \ldots, H_\ell \subseteq V$
2: compute $(\delta+1)$-coloring φ_i of each subgraph $G(H_i)$ in parallel
3: each node v of layer $i \in [\ell]$ chooses color $c(v) = (i, \varphi_i(v))$
4: run `Sparse_Set`(c, δ) on G to obtain set $X \subseteq V$
5: return X as a MWIS approximation

Overview of Algorithm 3. Algorithm 3 is very simple. First, `BE_Partition`(α, ϵ) (see Section 2) is invoked to obtain layers $H_1, \ldots, H_\ell$ (recall that $\ell = O(\log n)$). Then, a $(\delta+1)$-coloring $\varphi_i : H_i \to [\delta+1]$ is computed in every node-induced subgraph $G(H_i)$ in parallel. Each node $v \in H_i$ chooses the color $c(v) = (i, \varphi_i(v))$. Finally, `Sparse_Set`(c, δ) is invoked on G to compute the MWIS approximation $X \subseteq V$, where the ordering of colors required for the notion of β-bounded coloring is defined naturally as $c(v) = (i, \varphi_i(v)) > c(u) = (j, \varphi_j(u)) \iff (i > j) \vee (i = j \wedge \varphi_i(v) > \varphi_j(u))$.

We are now ready to prove Theorem 4.1.

Proof of Theorem 4.1. We start by analyzing the runtime. First, `BE_Partition` takes $O(\log n)$ time. For the $(\delta+1)$-coloring of all layers, notice that the maximum degree in each subgraph $G(H_i)$ is δ. Thus, we can employ an algorithm that computes a $(\Delta+1)$-coloring on a graph with maximum degree Δ. By [7], this can be done in time $O(\delta^{3/4}\log\delta + \log^* n) = O(\alpha^{3/4}\log\alpha + \log^* n)$. Finally, the number of colors assigned by the coloring c is $O(\delta \log n) = O(\alpha \log n)$. Thus, the call to `Sparse_Set`$(c, \delta)$ takes $O(\alpha \log n)$ time. Overall, the runtime of Algorithm 3 is $O(\alpha \log n)$.

Towards showing correctness, we shall show that c is a δ-bounded coloring. To that end, first observe that c is a proper coloring. Indeed, for an edge $(u, v) \in E$, if u and v are not in the same layer, then clearly $c(u) \neq c(v)$. Otherwise, let i be the index such that $u, v \in H_i$. By the correctness of the coloring performed on $G(H_i)$, we get that u and v were given different colors $\varphi_i(u) \neq \varphi_i(v)$ and thus, $c(u) \neq c(v)$.

Consider a node $v \in V$, let $i \in [\ell]$ be the index such that $v \in H_i$, and let $L(v) = \{u \in N(v) \mid c(u) > c(v)\}$. To see that c is δ-bounded, notice that by definition of c, it follows that $L(v) \subseteq \cup_{j=i}^{\ell} H_j$. Since v has at most δ neighbors in $\cup_{j=i}^{\ell} H_j$, we get that $|L(v)| \leq \delta$. The correctness of Algorithm 3 now follows directly from Lemma 3.10 ◀

We observe that replacing the invocation of `Sparse_Set` (line 4) with a call to Algorithm 2 results in the following theorem.

▶ **Theorem 4.2.** *There exists an algorithm that computes an $O(\alpha^2)$-approximation for MWIS in $O(\log n + \texttt{COL}(n, (2+\epsilon)\cdot\alpha) + \sqrt{\alpha \log n})$ rounds, where $\texttt{COL}(n, \Delta)$ is the runtime of $(\Delta+1)$-coloring an n-node graph with maximum degree Δ.*

For example, plugging the $(\Delta+1)$-coloring algorithm of [7] into Theorem 4.2 leads to a runtime bound of $O(\log n + \alpha^{3/4}\log\alpha + \sqrt{\alpha \log n})$.

4.2 Faster Algorithms

In this section we present a generic approximation algorithm (Algorithm 4) parameterized by an integer $k > 0$. Specifically, we show the following.

▶ **Lemma 4.3.** *For any integer $k > 0$, Algorithm 4 computes an $(8^k \cdot \alpha)$-approximation for MWIS. The runtime of Algorithm 4 is $O(k \cdot \alpha^{1/k} \cdot \log n)$.*

Later on in the section, we demonstrate the applicability of the generic algorithm (see Theorems 4.7,4.8,4.9, and 4.10). We now give an overview of the algorithm. Refer to Algorithm 4 for a pseudocode description.

Algorithm 4 A $(8^k \cdot \alpha)$-approximation algorithm for an α-arboricity graph $G = (V, E)$ and parameter $k \in \mathbb{Z}_{\geq 1}$.

$X_0 = V$
for $t = 0, \ldots, k-2$ **do**
  run `BE_Partition`$(\alpha^{(k-t)/k}, \epsilon = 1/100)$ on $G(X_t)$ to compute layers $H_1^t, \ldots, H_\ell^t$
  compute a $(\frac{1}{4}\alpha^{(k-t-1)/k})$-arbdefective $O(\alpha^{1/k})$-coloring φ_i on each $G(H_i^t)$
  each node v of layer $i \in [\ell]$ chooses the color $c_t(v) = (i, \varphi_i(v))$
  let $B_t = \{(u, v) \in E \mid u, v \in X_t,\ c_t(u) \neq c_t(v)\}$
  run `Sparse_Set`$(c_t, 4\alpha^{1/k})$ on subgraph (X_t, B_t) to obtain set $X_{t+1} \subseteq V$
run Algorithm 3 on $G(X_{k-1})$ with parameters $\alpha^{1/k}$ and $\epsilon = 1/100$ to obtain set $X_k \subseteq V$
return X_k as a MWIS approximation

Overview of Algorithm 4. The algorithm starts by performing $k-1$ phases $t = 0, \ldots, k-2$. Starting from $X_0 = V$, in each phase t, the set $X_{t+1} \subseteq X_t$ is computed. The goal is for X_{t+1} to not be "too far" from X_t in terms of weight while having a considerably sparser induced subgraph. To that end, we start by computing a coloring c_t as follows. First, `BE_Partition` is invoked on $G(X_t)$ to compute layers $H_1^t, \ldots, H_\ell^t$. Then, an arbdefective coloring φ_i that uses $O(\alpha^{1/k})$ colors is computed on each $G(H_i^t)$. The coloring c_t is obtained by taking $c_t(v) = (i, \varphi_i(v))$ for each node $v \in V$. The set X_{t+1} is then computed by an invocation of `Sparse_Set`$(c_t, 4\alpha^{1/k})$ on the subgraph (X_t, B_t), where $B_t = \{(u, v) \in E \mid u, v \in X_t,\ c_t(u) \neq c_t(v)\}$ is the set of bi-chromatic edges (with respect to c_t) in $G(X_t)$. After completing the $k-1$ phases, the output set X_k is computed by an invocation of Algorithm 3 on the subgraph $G(X_{k-1})$ (notice that in the case of $k = 1$, Algorithm 4 is just an invocation of Algorithm 3).

We now analyze Algorithm 4 starting from the following observation.

▶ **Observation 4.4.** *Every $0 \leq t \leq k-1$, satisfies the following: (1) the arboricity of $G(X_t)$ is at most $\alpha^{(k-t)/k}$; and (2) c_t is a $(3\alpha^{(k-t)/k})$-bounded coloring of the subgraph (X_t, B_t).*

Proof. We prove the assertion by induction over t. For the base case, consider $t = 0$. By definition, the arboricity of $G(X_0) = G$ is $\alpha = \alpha^{(k-0)/k}$. To see that c_0 is indeed a (3α)-bounded coloring of (X_0, B_0), first notice that by the definition of B_0, it holds that $c_0(u) \neq c_0(v)$ for every edge $(u, v) \in B_0$. That is, c_0 is a proper coloring of (X_0, B_0). Moreover, by the properties of `BE_Partition`, it follows that each node $v \in H_i^0$ has at most $(2 + 1/100)\alpha < 3\alpha$ neighbors in the layers $\cup_{j=i}^{\ell} H_j^0$. We can now establish that c_0 is (3α)-bounded since for each node $v \in H_i^0$, all of v's neighbors with larger color must be in $\cup_{j=i}^{\ell} H_j^0$.

Suppose now that the assertion holds for some $t \geq 0$. First, it is essential to show that the arbdefective coloring of line 4 is in fact computable. Recall that by [10], it is possible to compute a (Δ/p)-arbdefective coloring that uses $O(p)$ colors in graphs of maximum degree Δ. Notice that by the induction hypothesis, $G(X_t)$ has arboricity at most $\alpha^{(k-t)/k}$. Hence, by the properties of `BE_Partition`, each $G(H_i^t)$ has maximum degree $O(\alpha^{(k-t)/k})$. Now, for a fitting choice of $p = O(\alpha^{1/k})$, we get an arbdefective coloring with the desired parameters.

For the step of the induction, we start by showing that the arboricity of $G(X_{t+1})$ is at most $\alpha^{(k-t-1)/k}$. Let $B_t = \{(u,v) \in E \mid u,v \in X_t,\ c_t(u) \neq c_t(v)\}$ be the set of bi-chromatic edges (with respect to c_t) in $G(X_t)$. We partition the edges of $G(X_{t+1})$ into two disjoint sets $M = \{(u,v) \in E - B_t \mid u,v \in X_{t+1}\}$ and $B = \{(u,v) \in B_t \mid u,v \in X_{t+1}\}$.

Observe that by the definition of arbdefective coloring, the arboricity of (X_{t+1}, M) is at most $\frac{1}{4}\alpha^{(k-t-1)/k}$. As for the subgraph (X_{t+1}, B), first notice that by the induction hypothesis, c_t is a $(3\alpha^{(k-t)/k})$-bounded coloring of (X_t, B_t). Now, because X_{t+1} is obtained by an invocation of `Sparse_Set`$(c_t, 4\alpha^{1/k})$ on (X_t, B_t), it follows from Observation 3.9 that the arboricity of the graph (X_{t+1}, B) is bounded from above by $\frac{3\alpha^{(k-t)/k}}{4\alpha^{1/k}} = \frac{3}{4}\alpha^{(k-t-1)/k}$. Overall, we get that the arboricity of $G(X_{t+1})$ is at most the sum of the two arboricities which is bounded by $\frac{1}{4}\alpha^{(k-t-1)/k} + \frac{3}{4}\alpha^{(k-t-1)/k} = \alpha^{(k-t-1)/k}$.

We are left to show that c_{t+1} is a $(3\alpha^{(k-t-1)/k})$-bounded coloring of (X_{t+1}, B_{t+1}). As we have already shown, the arboricity of $G(X_{t+1})$ is at most $\alpha^{(k-t-1)/k}$. Hence, by similar arguments to the ones presented for the base of the induction, it follows that c_{t+1} is a $(3\alpha^{(k-t-1)/k})$-bounded coloring of (X_{t+1}, B_{t+1}). ◀

Towards bounding the approximation ratio, we make the following observation.

▶ **Observation 4.5.** *For every $0 \leq t \leq k-1$, it holds that $w(X_t) \leq 8\alpha^{1/k} \cdot w(X_{t+1})$.*

Proof. First, consider $t = k-1$. By Observation 4.4, the arboricity of $G(X_{k-1})$ is at most $\alpha^{1/k}$. Since we compute the set X_k by running Algorithm 3 on $G(X_{k-1})$, it follows from Theorem 4.1 that $w(X_{k-1}) \leq 2 \cdot (2+1/100) \cdot \alpha^{1/k} \cdot w(X_k) < 8\alpha^{1/k} \cdot w(X_k)$. Now, for $t < k-1$, notice that X_{t+1} is obtained by an invocation of `Sparse_Set`$(c_t, 4\alpha^{1/k})$. Thus, it follows from Lemma 3.1 that $w(X_t) \leq 2 \cdot 4\alpha^{1/k} \cdot w(X_{t+1}) = 8\alpha^{1/k} \cdot w(X_{t+1})$. ◀

As a consequence of Observation 4.5, we get the following lemma.

▶ **Lemma 4.6.** *$OPT(G) \leq 8^k \cdot \alpha \cdot w(X_k)$.*

Proof. It follows from Observation 4.5 that $w(X_0) \leq (8\alpha^{1/k})^k \cdot w(X_k) = 8^k \cdot \alpha \cdot w(X_k)$. The assertion follows since $OPT(G) \leq w(V) = w(X_0)$. ◀

We are now prepared to prove Lemma 4.3.

Proof of Lemma 4.3. The correctness follows directly from Lemma 4.6. Regarding the runtime, consider a phase $0 \leq t \leq k-2$. The invocation of `BE_Partition` in phase t takes $O(\log n)$ rounds. Then, computing an arbdefective coloring that uses $O(\alpha^{1/k})$ colors can be done in $O(\alpha^{1/k} + \log^* n)$ due to the algorithm of [10]. Notice that the coloring c_t uses $O(\alpha^{1/k} \cdot \ell) = O(\alpha^{1/k} \cdot \log n)$ colors. Thus, the runtime of `Sparse_Set`$(c_t, 4\alpha^{1/k})$ on subgraph (X_t, B_t) takes $O(\alpha^{1/k} \cdot \log n)$ rounds. This means that the total runtime of all $k-1$ phases is $O(k \cdot \alpha^{1/k} \cdot \log n)$. Finally, in line 8, Algorithm 3 is invoked on a graph with arboricity at most $\alpha^{1/k}$. As established in 4.1, this requires $O(\alpha^{1/k} \cdot \log n)$ rounds. Overall, we get that the runtime of Algorithm 4 is $O(k \cdot \alpha^{1/k} \cdot \log n)$. ◀

We turn to explore some new bounds that can be derived from Lemma 4.3.

▶ **Theorem 4.7.** *For any constant $\tau > 0$, there exists an $O(\alpha^\tau \log n)$-round algorithm that computes an $O(\alpha)$-approximation for MWIS.*

Proof. The algorithm is obtained by running Algorithm 4 with $k = \lceil 1/\tau \rceil$. The approximation ratio achieved is $8^k \cdot \alpha = 8^{\lceil 1/\tau \rceil} \cdot \alpha = O(\alpha)$. The runtime is $O(k \cdot \alpha^{1/k} \cdot \log n) = O(\alpha^\tau \log n)$. ◀

Observe that since $\alpha \leq \Delta$, the algorithm of Theorem 4.7 is also an $O(\Delta)$-approximation algorithm. As described in [21, Lemma 4.6] (based on the local-ratio approach of [31]), given an $O(\Delta)$-approximation algorithm $\mathcal{A}$ for MWIS, one can improve the approximation factor to $\Delta(1+\epsilon)$ for any parameter $\epsilon > 0$, at the cost of $O(\log(1/\epsilon))$ repetitions of $\mathcal{A}$. This leads to the following theorem.

▶ **Theorem 4.8.** *Let $\epsilon > 0$ be a parameter. For any constant $\tau > 0$, there exists an $O(\alpha^\tau \log n \log(1/\epsilon))$-round algorithm that computes a $\Delta(1+\epsilon)$-approximation for MWIS.*

Another direct consequence of Algorithm 4 is the following theorem.

▶ **Theorem 4.9.** *For any constant $\tau > 0$, there exists an $O(\log \alpha \log n)$-round algorithm that computes an $\alpha^{1+\tau}$-approximation for MWIS.*

Proof. The algorithm is obtained by running Algorithm 4 with $k = \lfloor \frac{\tau}{3} \log \alpha \rfloor$. The approximation ratio achieved is $8^k \cdot \alpha \leq 8^{\frac{\tau}{3} \log \alpha} \cdot \alpha = \alpha^\tau \cdot \alpha = \alpha^{1+\tau}$. The runtime is $O(k \cdot \alpha^{1/k} \cdot \log n) = O(\log \alpha \log n) \cdot 2^{O(1/\tau)} = O(\log \alpha \log n)$. ◀

The following theorem follows directly from Theorem 4.9.

▶ **Theorem 4.10.** *For any graph $G = (V, E)$ with arboricity α and maximum degree Δ such that $\alpha = \Delta^{1-\Theta(1)}$, there exists an algorithm that computes a $\Delta^{1-\Theta(1)}$-approximation for MWIS in $O(\log \alpha \log n)$ rounds.*

5 Approximation Algorithm for Directed Graphs with Bounded Out-Degree

Consider a directed graph $G = (V, E)$ where the out-degree of each node is bounded by an integer $d > 0$, and let $w : V \to \mathbb{R}_{\geq 0}$ be a node-weight function. In this section, we present an algorithm (Algorithm 5) that computes a $2d^2$-approximation for MWIS in G. More concretely, we prove the following theorem.

▶ **Theorem 5.1.** *For a directed graph $G = (V, E)$ with out-degree at most d, Algorithm 5 computes a $2d^2$-approximation for MWIS in time $O(d^2 + \log^* n)$.*

Algorithm 5 A $2d^2$-approximation algorithm for MWIS on a directed graph $G = (V, E)$ with out-degree at most d.

1: compute a proper $O(d^2)$-coloring c
2: let $E_{inc} = \{(v \to u) \in E \mid c(v) < c(u)\}$
3: run `Sparse_Set`(c, d) on $G_{inc} = (V, E_{inc})$ to obtain set $X \subseteq V$
4: run `Sparse_Set`$(-c, d)$ on $G(X)$ to obtain set $X' \subseteq V$
5: return X' as a MWIS approximation

Overview of Algorithm 5. First, Algorithm 5 computes a proper coloring $c : V \to [k]$ of G, such that $k = O(d^2)$. Then, let $E_{inc} = \{(v \to u) \in E \mid c(v) < c(u)\}$ be the set of edges in G directed towards the endpoint with larger color. Algorithm 5 then invokes `Sparse_Set`(c, d) on the graph $G_{inc} = (V, E_{inc})$ to compute a set X. Notice that X is not necessarily an independent set in G. To correct that, Algorithm 5 invokes `Sparse_Set`$(-c, d)$ again, this time on the node-induced subgraph $G(X)$ with the coloring $(-c)$ (defined simply as $-c(v) = (-1) \cdot c(v)$ for all $v \in V$) to obtain the set X' which is returned as a MWIS approximation.

We now analyze Algorithm 5 starting from the following two simple observations.

▶ **Observation 5.2.** *X' is an independent set in G.*

▶ **Observation 5.3.** *c is a d-bounded coloring of G_{inc}.*

Proof. Consider a node v. By construction, each neighbor u of v in G_{inc} that satisfies $c(u) > c(v)$, must be an outgoing neighbor. Since v has at most d outgoing neighbors, it follows that c is a d-bounded coloring. ◀

We now establish the following lemma regarding the weight of X.

▶ **Lemma 5.4.** $d \cdot w(X) \geq OPT(G)$

Proof. Let $OPT(G_{inc})$ be the weight of a MWIS in G_{inc}. Observation 5.3 combined with Lemma 3.10 implies that $d \cdot w(X) \geq OPT(G_{inc})$. To see that $OPT(G_{inc}) \geq OPT(G)$, observe that any independent set in G is also an independent set in G_{inc}. Therefore, we get that $d \cdot w(X) \geq OPT(G_{inc}) \geq OPT(G)$. ◀

We move on to bound the weight of X'. To that end, we make the following observation.

▶ **Observation 5.5.** *$(-c)$ is a d-bounded coloring of $G(X)$.*

Proof. Consider a node $v \in X$ and let $L'(v) = \{u \in X \cap N(v) \mid -c(v) < -c(u)\}$ be the set of v's neighbors in $G(X)$ that have a larger color assigned by the coloring $(-c)$. Notice that equivalently, $L'(v) = \{u \in X \cap N(v) \mid c(v) > c(u)\}$. Consider an incoming neighbor $u \in N(v)$ of v. If $c(v) > c(u)$, then $(u \to v) \in E_{inc}$. Since X is an independent set of G_{inc} and $v \in X$, it follows that $u \notin X$. Therefore, all nodes in $L'(v)$ are outgoing neighbors of v. Because v has at most d outgoing neighbors, we get that $|L'(v)| \leq d$ which concludes our proof. ◀

The following corollary follows directly from Observations 5.3 and 5.5, and Lemma 3.10.

▶ **Corollary 5.6.** $2d \cdot w(X') \geq w(X) \geq OPT(G)/d$

We can now prove Theorem 5.1.

Proof of Theorem 5.1. The correctness of Algorithm 5 follows directly from Observation 5.2 and Corollary 5.6. Regarding runtime, the coloring c can be computed in $O(\log^* n)$ rounds by means of a coloring algorithm by Linial [34].[5] Following that, c and $(-c)$ use $O(d^2)$ colors. Thus, each invocation of `Sparse_Set` takes $O(d^2)$ rounds. Overall, we get that the runtime of Algorithm 5 is $O(d^2 + \log^* n)$. ◀

[5] While Linial's algorithm computes an $O(\Delta^2)$-coloring in undirected graphs with maximum degree Δ, it can be applied to compute an $O(d^2)$-coloring in directed graph with out-degree at most d in a straightforward manner.

MWIS approximation requires $O(\log^* n)$ rounds. In [16], Czygrinow et al. prove the following lemma.[6]

▶ **Lemma 5.7** ([16]). *There is no deterministic distributed algorithm that finds an independent set of size $\Omega(n/\log^* n)$ in a cycle on n vertices in $o(\log^* n)$ rounds.*

We note that this lemma holds even in the LOCAL model, where the message size is not restricted. Moreover, the proof of Czygrinow et al. also applies for the case of oriented rings (where the edges are directed such that each node has out-degree 1). Since any n-node ring contains an independent set of size $\Omega(n)$, this lower bound implies that any algorithm that computes an $O(\log^* n)$-approximation for (unweighted) MaxIS in an oriented ring requires $\Omega(\log^* n)$ rounds. Therefore, the dependency on n in the runtime of Algorithm 5 cannot be improved.

References

1 Noga Alon and Nabil Kahalé. Approximating the independence number via the theta-function. *Math. Program.*, 80:253–264, 1998. doi:10.1007/BF01581168.

2 Saeed Akhoondian Amiri, Stefan Schmid, and Sebastian Siebertz. A local constant factor MDS approximation for bounded genus graphs. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 227–233. ACM, 2016. doi:10.1145/2933057.2933084.

3 Per Austrin, Subhash Khot, and Muli Safra. Inapproximability of vertex cover and independent set in bounded degree graphs. *Theory Comput.*, 7(1):27–43, 2011. doi:10.4086/TOC.2011.V007A003.

4 Nir Bachrach, Keren Censor-Hillel, Michal Dory, Yuval Efron, Dean Leitersdorf, and Ami Paz. Hardness of distributed optimization. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 – August 2, 2019*, pages 238–247. ACM, 2019. doi:10.1145/3293611.3331597.

5 Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. *J. ACM*, 48(5):1069–1090, 2001. doi:10.1145/502102.502107.

6 Reuven Bar-Yehuda, Keren Censor-Hillel, Mohsen Ghaffari, and Gregory Schwartzman. Distributed approximation of maximum independent set and maximum matching. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 165–174. ACM, 2017. doi:10.1145/3087801.3087806.

7 Leonid Barenboim. Deterministic $(\Delta + 1)$-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *J. ACM*, 63(5):47:1–47:22, 2016. doi:10.1145/2979675.

8 Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 410–419. ACM, 2010. doi:10.1145/1835698.1835797.

9 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distributed Comput.*, 22(5-6):363–379, 2010. doi:10.1007/S00446-009-0088-2.

10 Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\Delta + 1)$-coloring and applications. *J. ACM*, 69(1):5:1–5:26, 2022. doi:10.1145/3486625.

11 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.10.

[6] A similar bound is presented in [32] by Lenzen and Wattenhofer.

12 Siu On Chan. Approximation resistance from pairwise-independent subgroups. *J. ACM*, 63(3):27:1–27:32, 2016. doi:10.1145/2873054.

13 Yi-Jun Chang and Zeyong Li. The complexity of distributed approximation of packing and covering integer linear programs. *CoRR*, abs/2305.01324, 2023. doi:10.48550/ARXIV.2305.01324.

14 Andrzej Czygrinow, Michal Hanckowiak, and Edyta Szymanska. Fast distributed approximation algorithm for the maximum matching problem in bounded arboricity graphs. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 668–678. Springer, 2009. doi:10.1007/978-3-642-10631-6_68.

15 Andrzej Czygrinow, Michal Hanckowiak, and Wojciech Wawrzyniak. Distributed packing in planar graphs. In Friedhelm Meyer auf der Heide and Nir Shavit, editors, *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 55–61. ACM, 2008. doi:10.1145/1378533.1378541.

16 Andrzej Czygrinow, Michal Hanckowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, volume 5218, pages 78–92. Springer, 2008. doi:10.1007/978-3-540-87779-0_6.

17 Andrzej Czygrinow, Michal Hanckowiak, Wojciech Wawrzyniak, and Marcin Witkowski. Distributed CONGESTBC constant approximation of MDS in bounded genus graphs. *Theor. Comput. Sci.*, 757:1–10, 2019. doi:10.1016/J.TCS.2018.07.008.

18 Michal Dory, Mohsen Ghaffari, and Saeed Ilchi. Near-optimal distributed dominating set in bounded arboricity graphs. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25–29, 2022*, pages 292–300. ACM, 2022. doi:10.1145/3519270.3538437.

19 Yuval Efron, Ofer Grossman, and Seri Khoury. Beyond alice and bob: Improved inapproximability for maximum independent set in CONGEST. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 511–520. ACM, 2020. doi:10.1145/3382734.3405702.

20 Salwa Faour, Marc Fuchs, and Fabian Kuhn. Distributed CONGEST approximation of weighted vertex covers and matchings. 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 17:1–17:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.OPODIS.2021.17.

21 Salwa Faour, Mohsen Ghaffari, Christoph Grunau, Fabian Kuhn, and Václav Rozhon. Local distributed rounding: Generalized to mis, matching, set cover, and beyond. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 4409–4447. SIAM, 2023. doi:10.1137/1.9781611977554.CH168.

22 Uriel Feige. Approximating maximum clique by removing subgraphs. *SIAM J. Discret. Math.*, 18(2):219–225, 2004. doi:10.1137/S089548010240415X.

23 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 784–797. ACM, 2017. doi:10.1145/3055399.3055471.

24 Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discret. Math.*, 1(4):434–446, 1988. doi:10.1137/0401044.

25 Magnús M. Halldórsson. Approximations of independent sets in graphs. In Klaus Jansen and Dorit S. Hochbaum, editors, *Approximation Algorithms for Combinatorial Optimization, International Workshop APPROX'98, Aalborg, Denmark, July 18-19, 1998, Proceedings*, volume 1444 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1998. doi:10.1007/BFB0053959.

26 Magnús M. Halldórsson. Approximations of weighted independent set and hereditary subset problems. *J. Graph Algorithms Appl.*, 4(1):1–16, 2000. doi:10.7155/JGAA.00020.

27 Eran Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. *SIAM J. Comput.*, 31(5):1608–1623, 2002. doi:10.1137/S0097539700381097.

28 Johan Håstad. Clique is hard to approximate within $n^{1-epsilon}$. In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 627–636. IEEE Computer Society, 1996.

29 David R. Karger, Rajeev Motwani, and Madhu Sudan. Approximate graph coloring by semidefinite programming. *J. ACM*, 45(2):246–265, 1998. doi:10.1145/274787.274791.

30 Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

31 Ken-ichi Kawarabayashi, Seri Khoury, Aaron Schild, and Gregory Schwartzman. Improved distributed approximations for maximum independent set. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179, pages 35:1–35:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.DISC.2020.35.

32 Christoph Lenzen and Roger Wattenhofer. Leveraging linial's locality limit. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, volume 5218 of *Lecture Notes in Computer Science*, pages 394–407. Springer, 2008. doi:10.1007/978-3-540-87779-0_27.

33 Christoph Lenzen and Roger Wattenhofer. Minimum dominating set approximation in graphs of bounded arboricity. 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 510–524. Springer, 2010. doi:10.1007/978-3-642-15763-9_48.

34 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

35 Adir Morgan, Shay Solomon, and Nicole Wein. Algorithms for the minimum dominating set problem in bounded arboricity graphs: Simpler, faster, and combinatorial. 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 33:1–33:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.33.

36 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

37 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 350–363. ACM, 2020. doi:10.1145/3357713.3384298.

38 Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001. URL: http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-65367-7.

A Lifting the Knowledge of Arboricity Assumption

Recall that the algorithms presented in Section 4 assume that the nodes know the value of α. In this section, we show how this assumption can be lifted. First, we address the procedure `BE_Partition` that is assumed to take α as input. We describe a method for lifting the assumption on the knowledge of α while incurring an $O(\log \alpha)$ multiplicative overhead to the number of layers ℓ computed during the procedure.

The implementation of Procedure `BE_Partition` without knowledge of α is similar to the one presented in [9] with a small additional modification. The idea of [9] is to run $\lceil \log n \rceil + 1$ parallel executions of `BE_Partition`$(2^i, \epsilon)$ for $i = 0, 1, \ldots, \lceil \log n \rceil$. Each node then chooses the layer to which it was clustered in the execution that minimizes i. This leads to a partition $H_1, \ldots H_\ell$ of the nodes into $\ell = O(\log \alpha \log n)$ layers such that each node $v \in V$ belonging to layer H_j has at most 2δ neighbors in $\cup_{k=j}^{\ell} H_k$ (where $\delta = \lfloor (2+\epsilon)\alpha \rfloor$). A problem that arises in this case is that the approximation ratio of Algorithm 3 becomes 2δ as the coloring c obtained by the algorithm is 2δ-bounded.

To avoid increase to the approximation ratio, one can slightly tweak the parameters used in the runs of `BE_Partition`. Let $\gamma, \epsilon' > 0$ be two constants such that $(2+\epsilon')(1+\gamma) \leq 2+\epsilon$. Now, change the parallel runs to execute `BE_Partition`$((1+\gamma)^i, \epsilon')$ for $i = 0, \ldots, \lceil \log_{1+\gamma} n \rceil$. In the resulting partition $H_1, \ldots, H_\ell$, each node $v \in H_j$ has at most δ neighbors in $\cup_{k=j}^{\ell} H_k$. Moreover, notice that each node $v \in V$ obtains an estimate $\alpha(v)$ which is bounded from above by α. We note that in our algorithms, whenever α is used, each node internally can simply use the estimate $\alpha(v)$. This modification does not affect the correctness.

As noted before, when invoking `BE_Partition` without knowledge of α, the number of layers becomes $\ell = O(\log \alpha \log n)$. As a consequence, the number of colors in each coloring computed during our algorithms increases by an $O(\log \alpha)$ factor. Since the runtime of `Sparse_Set` depends on the number of colors, we get that apart from Theorem 4.2, the upper bound on all runtimes stated in Section 4 is multiplied by $O(\log \alpha)$; whereas the runtime of the $O(\alpha^2)$-approximation algorithm stated in Theorem 4.2 changes to $O(\log n + \texttt{COL}(n, (2+\epsilon)\cdot\alpha) + \sqrt{\alpha \log \alpha \log n})$.

A Wait-Free Deque With Polylogarithmic Step Complexity

Shalom M. Asbell
Department of Electrical Engineering and Computer Science, York University, Toronto, Canada

Eric Ruppert
Department of Electrical Engineering and Computer Science, York University, Toronto, Canada

Abstract

The amortized step complexity of operations on all previous lock-free implementations of double-ended queues is linear in the number of processes. This paper presents the first concurrent double-ended queue where the amortized step complexity of each operation is polylogarithmic. Since a stack is a special case of a double-ended queue, this is also the first concurrent stack with polylogarithmic step complexity. The implementation is wait-free and the amortized step complexity is $O(\log^2 p + \log q)$ per operation, where p is the number of processes and q is the size of the double-ended queue.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Concurrent algorithms

Keywords and phrases Lock-Free, Wait-Free, Double-Ended Queue, Deque, Stack, Space-Bounded, Polylogarithmic, Linearizable

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.17

Funding This research was funded by the Natural Sciences and Engineering Research Council of Canada and a Lassonde Undergraduate Research Award.

Acknowledgements We thank the anonymous reviewers for their detailed comments. This research was funded by the Natural Sciences and Engineering Research Council of Canada and a Lassonde Undergraduate Research Award.

1 Introduction

This paper describes a linearizable, wait-free implementation of a double-ended queue (deque) [20], which allows items to be added or removed at both ends. As an example of the concurrent deque's ubiquity, the class `ConcurrentLinkedDeque` is part of the Java standard library. Unlike previous lock-free deques, the amortized number of steps per operation in ours is polylogarithmic. To achieve this, we build on the recent wait-free queue of Naderibeni and Ruppert (N&R) [26], which was the first lock-free queue with polylogarithmic step complexity. All prior lock-free queues shared by p processes had $\Omega(p)$ amortized step complexity due to what Morrison and Afek called the *CAS retry problem* [25], which occurs when all contending update operations must repeatedly perform compare-and-swap (CAS) instructions on one location until they succeed. A successful CAS may thwart an attempt of each other process, yielding the $\Omega(p)$ amortized bound. Moreover, operations may starve.

To avoid the CAS retry problem and achieve polylogarithmic step complexity, the N&R queue uses a helping technique that was introduced by Afek, Dauber and Touitou [1] and used in Jayanti and Petrovic's single-dequeuer queue [18]. Each process is assigned a leaf in a static binary tree called an *ordering tree*. A process inserts its operation *op* into its leaf and then propagates *op* to the tree's root. At each node along the path to the root, *op* collects non-propagated operations from the node's children and attempts to CAS the *set* of all these operations into the node. If *op* fails its CAS on a node twice, some other process must have

propagated *op* to this node, so *op* can continue up the tree. Operations are linearized in the order they reach the tree's root. A key innovation of the N&R queue is their implicit representation of sets of operations designed so that sets can be efficiently collected and propagated, while still supporting efficient retrieval of each dequeue's result.

Since it is unclear that the implicit representation of sets of operations in the N&R queue can be extended to deques (or stacks), we design a very different explicit representation that stores the sets of operations in balanced search trees. In addition to allowing us to gather and propagate sets of operations up the tree quickly, our new representation allows us to apply these batches of operations when they reach the root to obtain an explicit representation of the current state of the deque. As a byproduct, we also get an explicit representation of the results of all dequeue operations in a batch of operations. This contrasts with the implicit representation used by the N&R queue, where the result of a dequeue had to be reconstructed by searching through the history of all operations applied to the queue.

Using our new representation, we obtain a wait-free, linearizable deque for p processes with amortized step complexity $O(\log^2 p + \log q)$ per operation, where q is the number of items in the deque. This matches the step complexity of the more restricted queue of N&R and is the first time sublinear amortized step complexity has been obtained for a lock-free deque (or stack, which is a restricted form of a deque). We unlink unneeded objects from our data structure to ensure it does not grow too large, which would slow operations down. However, we leave the orthogonal problem of reclaiming memory of unlinked items to a garbage collector, such as the highly optimized one provided by Java. Our ordering tree data structure uses $O(q + p^2 \log p)$ space, improving on the N&R queue by a factor of p.

2 Related Work

Deques. The deque is a classical data structure [20], and lock-free implementations of deques have been studied for decades. Although it is a fairly simple data structure, implementing it in a concurrent setting can be tricky. For example, Koval et al. [22] recently reported a linearizability bug in the `ConcurrentLinkedDequeue` of the Java standard library. Early lock-free deques [2, 6, 10, 23] used double-word CAS instructions, which are generally not provided by hardware. Michael [24] introduced the first lock-free deque based on single-word CAS. It uses a doubly-linked list. Each operation uses a CAS to update an *Anchor* object, which has pointers to both ends of the list. If the CAS fails, the operation tries again. This CAS retry problem yields $\Omega(p)$ amortized step complexity per operation, where p is the number of processes. Ideally, operations on opposite ends of the deque should not interfere with each other, but they do in the Michael deque since they are serialized by their accesses to the Anchor. Sundell and Tsigas [28] gave a lock-free deque, also based on a doubly-linked list, where operations at opposite ends of the deque do not interfere with each other, unless the deque is empty or nearly empty. This deque also suffers from the CAS retry problem.

Herlihy, Luchangco and Moir [16] gave an array-based deque satisfying the weaker progress condition of obstruction-freedom, where operations terminate if they run without interference from other processes for sufficiently long. Graichen, Izraelevitz and Scott [9] described how to use a doubly-linked list of these array-based deques to get an obstruction-free deque with unlimited capacity. Both implementations have unbounded amortized step complexity.

Restricted Deques. Both FIFO queues and LIFO stacks are restricted versions of deques: each provides only two of the four deque operations. Naderibeni and Ruppert [26] survey the extensive previous literature on lock-free queues, all having amortized step complexity $\Omega(p)$, before providing their wait-free queue with polylogarithmic step complexity. An

unbounded-space version of their queue achieves a step complexity of $O(\log p)$ per enqueue and $O(\log^2 p + \log q)$ per dequeue, where q is the size of the queue. A space-bounded version of their queue has amortized step complexity of $O(\log p \log(p+q))$ per operation.

Treiber [30] described a lock-free stack based on a singly-linked list with a pointer to the top element. Pushes and pops do a CAS on this pointer and retry if the CAS fails. It has $\Omega(p)$ amortized step complexity due to the CAS retry problem. Array-based stacks (e.g., [27]) also suffer from the CAS retry problem. Hendler, Shavit and Yerushalmi [14] used *elimination* to improve the performance of stacks: a concurrent push and pop can eliminate each other, with the pop returning the argument of the push. Operations access a traditional list- or array-based stack only if they fail to find a partner operation to eliminate. However, if pushes are only concurrent with other pushes, accesses to that underlying stack still take $\Omega(p)$ steps. Dodds, Haas and Kirsch's [7] lock-free stack assigns timestamps to pushed elements. Timestamps provide a partial order on the elements: elements added by concurrent pushes need not be ordered. Each process maintains a list of elements it has pushed, and a pop must check each of these lists to find an element with the youngest timestamp, which requires $\Omega(p)$ steps in the worst case. Pushes are artificially slowed down to make elimination more likely, thereby improving performance. The authors also sketch how a similar approach could be used for a queue or deque (in which dequeue operations would also require $\Omega(p)$ steps). Haas's thesis [12] gives pseudocode for the queue and deque, but says the linearization proof for the deque "is still a work in progress".

There are also lock-free deques where some operations are restricted to certain processes. For example, Arora, Blumofe and Plaxton [3] designed a lock-free deque for their work-stealing algorithm. In their deque, only one process can access one end, and only dequeues can be performed at the other end. It uses a fixed-size array and therefore has bounded capacity. Hendler et al. [13] used a doubly-linked list of arrays to remove this limitation. In both deques, the CAS retry problem occurs at the end of the queue that supports concurrent accesses: one successful dequeue at that end can cause all concurrent dequeues to fail.

Universal Constructions. A *universal construction* [15] is a general technique for constructing a lock-free implementation of any data structure. However, the resulting implementation is typically much less efficient than hand-crafted ones. Afek, Dauber and Touitou's universal construction [1] can achieve $O(\log p)$ steps per operation, as observed by Jayanti [17], but only if memory words can store $\Omega(p \log p)$ bits. With more reasonably sized $O(\log p)$-bit words, the universal construction would take $\Omega(p \log p)$ steps per operation. Nevertheless, the technique introduced in this construction forms the basis of our deque design.

Lower Bounds. Attiya and Fouren [4] gave lower bounds in terms of contention c, the number of operations that run concurrently. They showed the amortized step complexity of a lock-free stack or queue must be $\Omega(\min(c, \log\log p))$. Jayanti, Tarjan and Boix-Adserà [19] showed the amortized step complexity of any stack or queue implementation is $\Omega(\log p)$.

Red-Black Trees. Our algorithm uses a classical search tree data structure. A *red-black tree* (RBT) [11] is a balanced binary search tree that can store a set of items sorted by key values. Insertions, deletions and searches on a RBT of n items can be done in $O(\log n)$ time. In addition, we make use of $O(\log n)$-time algorithms for splitting and joining RBTs [29]. As in an order-statistic tree [5], an RBT can be augmented with a count of the number of nodes in the subtree rooted at that node, so that we can also select the ith element in an in-order traversal of the tree, or split the tree at that ith element, in $O(\log n)$ time. We assume a purely functional implementation of RBTs for a single process. In other words, we

assume update operations do not destroy the original version of the RBT, but rather create a new, modified version. This persistence can be achieved through *path copying* [8]: each RBT operation makes a copy of any node it visits, and modifies that copy without overwriting the tree's original state. In our deque algorithm, a process that wishes to update a shared RBT creates a new copy of the RBT using path copying and then swings a pointer from the old root to the new root. Processes essentially get a snapshot of the RBT simply by reading the pointer to the root, since all RBT nodes are immutable. Other persistent search trees (such as B-trees or AVL trees) could be used in place of RBTs.

3 Implementation

Our deque uses an *ordering tree*, which is a static binary tree of height $\lceil \log_2 p \rceil$ with one leaf assigned to each process. Each node stores a sequence of *Blocks*; each Block represents a set of concurrent deque operations. The four deque operations TopEnqueue, TopDequeue, BotEnqueue and BotDequeue allow items to be enqueued or dequeued from either end of the deque, which we call top and bottom to avoid confusion with the right and left directions used to describe children in the ordering tree. To perform an operation on the deque, a process inserts a Block containing just that operation into its own leaf, and then ensures that the operation is propagated up to the root. Processes help propagate one another's operations along with their own. Operations are linearized when they reach the root. To propagate its operation, a process performs a *double refresh* at each node along the path from its leaf to the root. A refresh at a node v creates a Block that contains any unpropagated operations in v's children and attempts to insert the Block into v using a CAS instruction. If the process fails this CAS twice, then it is guaranteed that some other process has propagated its operation to v. So far, this overall approach is similar to previous constructions [1, 18, 26].

In the N&R queue [26], each internal node does not have an *explicit* representation of the operations that have reached the node. More specifically, an item that is enqueued is stored in the leaf of the process that enqueued it, but is not stored in any of the internal nodes. Instead, each Block in an internal node stores some metadata that serves as an implicit representation of its set of operations. This metadata is used to navigate to an operation's Block in each node along the root-to-leaf path. Processes can compute the response for each dequeue by navigating these paths (both up and down the tree). Because each Block can contain up to p operations, Naderibeni and Ruppert claimed that building an explicit representation of the operations in Blocks of internal nodes would be too costly to achieve polylogarithmic step complexity. However, our new representation shows that, in fact, we can create such explicit representations without sacrificing the polylogarithmic running time. Moreover, this explicit representation is essential for our deque. In a queue, the FIFO ordering makes it easy for a dequeue to find its response using very little information about the sequence of operations: the ith (non-empty) dequeue returns the argument of the ith enqueue. Since a deque (or stack) permits LIFO access, matching dequeues to enqueues is less straightforward. The explicit state of our deque allows us to determine this matching.

The key idea is to use *red-black trees* (RBTs) to represent the state of the deque and batches of items to be enqueued into the deque. We use *persistent* RBTs, meaning that update operations are non-destructive because they create a new copies of RBT nodes and modify those copies, leaving the original nodes unchanged. Since RBT nodes are immutable, reading a pointer to an RBT's root gives us a snapshot of the RBT. At the root of the ordering tree, we store a *state* RBT whose in-order traversal gives the items in the implemented deque from bottom to top. In each node v of the ordering tree, we store an RBT called *topEnqs*

whose in-order traversal is the sequence of values enqueued by all TopEnqueues that have been propagated to the node, in the order they reached v. Another such RBT *botEnqs* is used for BotEnqueues. This use of RBTs allows us to perform five key actions efficiently:

1. gather the batch of items enqueued by several consecutive Blocks of a node v by splitting off the required number of most recently added items from v's *topEnqs* or *botEnqs* RBT,
2. add such a batch to v's parent's *topEnqs* RBT (or *botEnqs* RBT) during a refresh by joining two RBTs,
3. apply a batch of enqueues to the state of the deque when they reach the root by joining the *state* RBT with the RBT representing the batch of enqueued items,
4. apply a batch of dequeues to the state of the deque when they reach the root by splitting the the required number of items off one side of the *state* RBT, and
5. discard old information when it is no longer needed (again by splitting an RBT) to avoid having RBTs that grow too large, which in turn would slow down operations on RBTs.

We linearize operations on our deque based on the order in which they were propagated to the root and applied to *state*. Because other operations may help propagate a dequeue to the root, a dequeue has to retrace the path it took from its leaf to the root to retrieve its response. When dequeues are linearized, RBTs of the items split off of the *state* RBT are saved in the root for later retrieval by those dequeues.

The Blocks of a node are also stored in an RBT called *blocks*. This makes it easy to garbage-collect old Blocks that are no longer needed by splitting the RBT. Each Block stores some metadata to describe its batch of operations. For example, this allows us to find the portion of the *topEnqs* tree that corresponds to TopEnqueues within a Block. When a Block is added to a node's *blocks* RBT, we want to ensure that the items of all enqueue operations represented by that Block are simultaneously added to the *topEnqs* and *botEnqs* RBTs of the node. For this reason, we actually store pointers to the roots of all of the node's RBTs in an object that we call the *rbts* tuple of the node. Thus, we can update all of a node's RBTs atomically by writing a pointer to this tuple in the node's *rbts* field. Likewise, we can get a consistent snapshot of all the node's RBTs by simply reading the node's *rbts* field.

3.1 Data Structure Description

We now describe the objects of our data structure in more detail. Nodes in our static *ordering tree* contain the following fields.

- *parent* – a pointer to the node's parent node.
- *left* – a pointer to the node's left child.
- *right* – a pointer to the node's right child.
- *rbts* – a pointer to a tuple of (pointers to the roots of) RBTs, described below.

In non-root nodes, the *rbts* tuple has the following five fields.

- *blocks* – an RBT composed of Block objects that each represent a set of concurrent operations. This tree is sorted by the order the Blocks were added to the node. Initially, *blocks* contains a single dummy Block whose fields are all 0.
- *topEnqs* and *botEnqs* – RBTs that store items to be enqueued at the top and bottom ends, respectively, of the deque. These trees are initially empty. When a Block of operations is added to *blocks*, the enqueued items of those operations are added to to the right end of these enqueue trees. Thus, the items added to the enqueue trees are all items enqueued by operations in Blocks that have been added to the *blocks* RBT, in the order the operations were added to *blocks*. Garbage collection will remove from the left end of these enqueue trees those items that are no longer needed because they have been propagated up to the enqueue trees in the parent node.

- $discarded_{topEnqs}$ and $discarded_{botEnqs}$ – integers that store the number of items that have been discarded from the left side of the *topEnqs* and *botEnqs* RBTs, respectively, by garbage collection.

In the root, the *rbts* tuple has the following two fields.

- *blocks* – defined the same way as for non-root nodes.
- *state* – an RBT storing the items in the deque after all the blocks of operations that have reached the root have been performed. The in-order traversal of the *state* RBT gives the order of the items from the bottom of the deque to the top. It is initially empty.

Each Block object in a node's *blocks* RBT has the following five fields.

- *index* – the number of Blocks that were added to the node's *blocks* RBT before this one.
- $sum_{topEnqs}$, $sum_{botEnqs}$, $sum_{topDeqs}$ and $sum_{botDeqs}$ – the total number of operations of each type that have been propagated to the node up to and including the current Block.

Blocks in internal nodes have the following two additional fields.

- end_{left} – index of the last Block propagated from the node's left child into this Block.
- end_{right} – index of the last Block propagated from the node's right child into this Block.

Each Block object in the root has the following two additional fields.

- *topDeqs* and *botDeqs* – RBTs containing the responses to be returned to the TopDequeues or BotDequeues, respectively, contained in the Block. The in-order traversals of *topDeqs* and *botDeqs* in a root Block B gives the order in which elements were dequeued from the top and bottom of the deque respectively.

Each Block in a leaf node has the following additional field.

- *response* – the response to a leaf Block's operation, if it is dequeue.

Most fields of objects are *immutable*, except for the *rbts* field of nodes and the *response* field in leaf Blocks. That is, after a process creates an object, it sets the values of all its immutable fields before writing a pointer to that object in shared memory, and those fields' values never change thereafter. Even the persistent red-black trees within our data structure have immutable nodes, as described above. This simplifies reasoning about concurrency in our algorithm, because a process can essentially obtain a snapshot of the *rbts* field of a node v and all of the information contained within it, including the information inside v's Blocks, simply by reading $v.rbts$. Thus, when proving correctness, we can often focus on the few lines of code where nodes' *rbts* fields are read or updated.

We assume RBTs have **Split** and **Join** operations [29]. Join(RBT l, RBT r) returns an RBT whose in-order traversal contains the elements of l's in-order traversal followed by r's in-order traversal. The inverse operation Split(RBT t, int *count*) returns a pair of RBTs $\langle l, r\rangle$ such that Join(l, r) would yield t, and where r contains *count* objects (or $t.size$ items if $count > t.size$) and l has the rest. The enqueue and dequeue RBTs have the additional operations **Append** and **SearchByRank**. Append(RBT t, Object e) returns the RBT that is produced by inserting e into t as the rightmost element. SearchByRank(RBT t, int r) returns the rth item in the in-order traversal of t. The *blocks* RBTs are sorted by the *index* field of the Blocks and also have **Insert**, **MaxBlock** and **SplitByIndex** operations. Insert(RBT t, Block B) inserts B into t. MaxBlock(RBT t) returns the Block with the highest index in t. Similar to a Split operation, SplitByIndex(RBT *blocks*, int i) returns a pair of RBTs $\langle l, r\rangle$, where l contains the Blocks with indices less than i and r contains the rest.

3.2 Pseudocode Description

Pseudocode for our implementation appears in Algorithms 1 to 3. We omit BotEnqueue, BotDequeue, GetBotEnqs, CompleteBotDeq and IndexBotDeq, which are identical to the corresponding routines for the top end of the deque, except for replacing all occurrences

of *top* by *bot*, and making the changes noted in comments on lines 6, 17 and 87. We use $blocks[i]$ as a shorthand for the Block with index i that was inserted into the *blocks* RBT: when the code reads $blocks[i]$ (e.g., on line 71), it does a BST search for *index* i in *blocks*.

To enqueue an item e at the top end of the deque, a process calls **TopEnqueue**(e). It creates a new Block B to represent the TopEnqueue in the process's leaf. B's *index* is one higher than the previous Block, and its sum_{topEnq} is one higher than the previous Block. The call to GC at line 7 makes a copy $rbts_{new}$ of the *rbts* field of the leaf and may perform garbage collection (described in Section 3.4) to discard obsolete information. Line 8 inserts B into $rbts_{new}.blocks$ and line 9 appends e to the right end of $rbts_{new}.topEnqs$. Then, line 10 writes $rbts_{new}$ into the process's leaf. Finally, line 11 calls Propagate to propagate the new operation to the root, thereby ensuring that it is linearized.

A **TopDequeue** follows a similar pattern. It creates a new Block B whose *index* and $sum_{topDeqs}$ fields are one higher than the previous Block in the process's leaf. It creates a new copy $rbts_{new}$ of the leaf's *rbts* field, possibly discarding unneeded information in GC, appends B to $rbts_{new}.blocks$ (lines 13–19), and writes $rbts_{new}$ into the process's leaf (line 20). TopDequeue then calls Propagate at line 21 to propagate the new operation upward, and finally calls CompleteTopDeq (described below) at line 22 to find the TopDequeue's response. This CompleteTopDeq may return *null*, but if this happens then another process helping the TopDequeue must have written the TopDequeue's response into $B.response$.

The **Propagate** method recursively propagates operations from leaves of the *ordering tree* to its root. Propagate calls Refresh at most twice on each internal node v along the path from the process's leaf to the root. As in previous work [1, 18, 26], this suffices to ensure that a Block containing the operation is added to the root.

To propagate the non-propagated Blocks in v's children along with their enqueued items to v, an operation calls **Refresh**(v). Refresh(v) constructs a new *rbts* tuple for v that includes the non-propagated Blocks of v's children and their enqueued items, and attempts to CAS the tuple into $v.rbts$ (line 51). To do this, it first reads snapshots of $v.rbts$, $v.left.rbts$ and $v.right.rbts$ into $rbts_{old}$, $rbts_{left}$ and $rbts_{right}$ (lines 28–30). Line 31 ensures that Refresh has a consistent view of v and its children; otherwise the Refresh aborts and returns false. Line 34 calls CreateBlock (described below) to construct a Block B_{new} with an index one higher than its preceding Block in v, B_{prev}. B_{new} contains the metadata of the non-propagated operations from $rbts_{left}$ and $rbts_{right}$. If there are no non-propagated operations in $rbts_{left}$ and $rbts_{right}$, then CreateBlock returns *null* at line 64 and Refresh returns true at line 35 because another Refresh already propagated operations from v's children into v.

If B_{new} is non-empty, line 36 of Refresh performs GC on $rbts_{old}$, which may discard unneeded Blocks and enqueued items in $rbts_{old}$ (see Section 3.4), and saves the updated tuple in $rbts_{new}$. Refresh then inserts B_{new} into $rbts_{new}.blocks$, retrieves the enqueued items corresponding to the enqueues in B_{new} using GetTopEnqs and GetBotEnqs (described below), and stores them in $newTopEnqs$ and $newBotEnqs$, respectively (lines 39–40). If v is not the root, Refresh joins $newTopEnqs$ and $newBotEnqs$ to the right ends of $rbts_{new}.topEnqs$ and $rbts_{new}.botEnqs$ (lines 42–43) so that the in-order of these trees represent enqueued items propagated to v in the order they reached v. Finally, line 51 attempts to CAS $rbts_{new}$ into $v.rbts$, returning the result of the CAS to indicate whether it succeeded.

If v is the root, Refresh instead performs the concurrent batch of operations represented by B_{new} on $rbts_{old}.state$ to obtain the new *state* of the deque, $rbts_{new}.state$. To do so, Refresh first calculates $num_{topDeqs}$ and $num_{botDeqs}$, the numbers of TopDequeues and BotDequeues it must perform on the *state*, using the *sum* fields in B_{new} and B_{prev} (lines 45–46). Refresh then does a batch of concurrent TopDequeues by Splitting $num_{topDeqs}$ items off of the right

Algorithm 1 Implementation of deque: main routines.

```
1:  TopEnqueue(Object e)                                  ▷ enqueue e to the top of the deque
2:      B' ← MaxBlock(leaf.rbts.blocks)                   ▷ previous Block in process's leaf
3:      let B be a new Block object with fields: index ← B'.index + 1,
4:          sum_topDeqs ← B'.sum_topDeqs, sum_botDeqs ← B'.sum_botDeqs
5:          sum_topEnqs ← B'.sum_topEnqs + 1,             ▷ increment sum_topEnqs for top enqueue
6:          sum_botEnqs ← B'.sum_botEnqs,                 ▷ BotEnqueue increments this instead
7:      rbts_new ← GC(leaf, leaf.rbts, B, B')             ▷ returns new rbts (after GC if necessary)
8:      rbts_new.blocks ← Insert(rbts_new.blocks, B)
9:      rbts_new.topEnqs ← Append(rbts_new.topEnqs, e)    ▷ insert e as rightmost element
10:     leaf.rbts ← rbts_new                              ▷ write new tuple into process's leaf
11:     Propagate(leaf.parent)                            ▷ propagate enqueue to root

12: TopDequeue : Object                                   ▷ dequeue item from top of deque
13:     B' ← MaxBlock(leaf.rbts.blocks)                   ▷ previous Block in process's leaf
14:     let B be a new Block object with fields: index ← B'.index + 1,
15:         sum_topEnqs ← B'.sum_topEnqs, sum_botEnqs ← B'.sum_botEnqs,
16:         sum_topDeqs ← B'.sum_topDeqs + 1,             ▷ increment sum_topDeqs for top dequeue
17:         sum_botDeqs ← B'.sum_botDeqs                  ▷ BotDequeue increments this instead
18:     rbts_new ← GC(leaf, leaf.rbts, B, B')             ▷ returns new rbts (after GC if necessary)
19:     rbts_new.blocks ← Insert(rbts_new.blocks, B)
20:     leaf.rbts ← rbts_new                              ▷ write new tuple into process's leaf
21:     Propagate(leaf.parent)                            ▷ propagate the operation to the root
22:     response ← CompleteTopDeq(leaf, B.index)          ▷ retrieve dequeue response
23:     return (response = null ? B.response : response)

24: Propagate(Node v) ▷ propagate operations from v's children up to root
25:     if not Refresh(v) then Refresh(v)                 ▷ double Refresh on v
26:     if v ≠ root then Propagate(v.parent)              ▷ recurse to parent

27: Refresh(Node v) : boolean                             ▷ propagate operations into v
28:     rbts_old ← v.rbts
29:     rbts_left ← v.left.rbts
30:     rbts_right ← v.right.rbts
31:     if v.rbts ≠ rbts_old then return false
32:     B_prev ← MaxBlock(rbts_old.blocks)                ▷ previous Block in v
33:     ▷ create a new Block B_new from non-propagated blocks in v.left and v.right
34:     B_new ← CreateBlock(v, B_prev, rbts_left, rbts_right)
35:     if B_new = null then return true                  ▷ no new operations to propagate into v
36:     rbts_new ← GC(v, rbts_old, B_new, B_prev)
37:     rbts_new.blocks ← Insert(rbts_new.blocks, B_new)  ▷ insert new block
38:     ▷ get the enqueued items from children that have not previously been propagated to v
39:     newTopEnqs ← GetTopEnqs(B_prev, B_new, rbts_left, rbts_right)
40:     newBotEnqs ← GetBotEnqs(B_prev, B_new, rbts_left, rbts_right)
41:     if v ≠ root then                                  ▷ join new items to enqueue RBTs
42:         rbts_new.topEnqs ← Join(rbts_old.topEnqs, newTopEnqs)
43:         rbts_new.botEnqs ← Join(rbts_old.botEnqs, newBotEnqs)
44:     else                                              ▷ apply operations on root's state
45:         num_topDeqs ← B_new.sum_topDeqs − B_prev.sum_topDeqs
46:         num_botDeqs ← B_new.sum_botDeqs − B_prev.sum_botDeqs
47:         ⟨newstate, B_new.topDeqs⟩ ← Split(rbts_old.state, num_topDeqs)
48:         ⟨B_new.botDeqs, newstate⟩ ← Split(newstate, newstate.size − num_botDeqs)
49:         newstate ← Join(newstate, newTopEnqs)
50:         rbts_new.state ← Join(newBotEnqs, newstate)
51:     return CAS(v.rbts, rbts_old, rbts_new)            ▷ CAS new rbts tuple into v
```

Algorithm 2 Implementation of deque: helper routines.

52: CreateBlock(Node v, Block B_{prev}, rbts $rbts_{left}$, rbts $rbts_{right}$) : Block
53: ▷ create new Block B from subblocks in $v.left$ and $v.right$
54: $B_L \leftarrow$ MaxBlock($rbts_{left}.blocks$)
55: $B_R \leftarrow$ MaxBlock($rbts_{right}.blocks$)
56: ▷ create a new Block B from non-refreshed $blocks$ in $v.left$ and $v.right$
57: let B be a new Block object with fields: $index \leftarrow B_{prev}.index + 1$,
58: $end_{left} \leftarrow B_L.index$, $end_{right} \leftarrow B_R.index$, ▷ new ends are max blocks in children
59: $sum_{topEnqs} \leftarrow B_L.sum_{topEnqs} + B_R.sum_{topEnqs}$,
60: $sum_{topDeqs} \leftarrow B_L.sum_{topDeqs} + B_R.sum_{topDeqs}$,
61: $sum_{botEnqs} \leftarrow B_L.sum_{botEnqs} + B_R.sum_{botEnqs}$,
62: $sum_{botDeqs} \leftarrow B_L.sum_{botDeqs} + B_R.sum_{botDeqs}$
63: **if** $B.end_{left} = B_{prev}.end_{left}$ **and** $B.end_{right} = B_{prev}.end_{right}$ **then**
64: **return** *null* ▷ B is empty (it has no subblocks)
65: **else return** B

66: GetTopEnqs(Block B_{prev}, Block B_{new}, rbts $rbts_{left}$, rbts $rbts_{right}$) : RBT
67: ▷ retrieve top enqueue items belonging to a new Block B_{new} from children
68: $blocks_L \leftarrow rbts_{left}.blocks$
69: $blocks_R \leftarrow rbts_{right}.blocks$
70: ▷ calculate number of newly promoted enqueues
71: $leftEnqs \leftarrow blocks_L[B_{new}.end_{left}].sum_{topEnqs} - blocks_L[B_{prev}.end_{left}].sum_{topEnqs}$
72: $rightEnqs \leftarrow blocks_R[B_{new}.end_{right}].sum_{topEnqs} - blocks_R[B_{prev}.end_{right}].sum_{topEnqs}$
73: ▷ split the required number of items from right ends of the children's $topEnqs$ RBTs
74: $\langle *, leftChunk\rangle \leftarrow$ Split($rbts_{left}.topEnqs$, $leftEnqs$)
75: $\langle *, rightChunk\rangle \leftarrow$ Split($rbts_{right}.topEnqs$, $rightEnqs$)
76: **return** Join($leftChunk$, $rightChunk$) ▷ join the child chunks left to right

77: CompleteTopDeq(Node $leaf$, int h) : Object ▷ return response of TopDequeue in Block h of $leaf$
78: ▷ after it has propagated to root, or *null* if helper has already recorded response.
79: ▷ first, find Block index of the dequeue in the root and its position within the block
80: $\langle b, i\rangle \leftarrow$ IndexTopDeq($leaf, h, 1$)
81: ▷ find response to the top dequeue according to its position in the root
82: **if** $\langle b, i\rangle \neq \langle 0, 0\rangle$ **then**
83: $B \leftarrow root.rbts.blocks[b]$ ▷ root Block containing dequeue
84: **if** $B \neq null$ **then**
85: **if** $B.topDeqs.size < i$ **then return** empty
86: **else return** SearchByRank($B.topDeqs, B.topDeqs.size - i + 1$)
87: ▷ in CompleteBotDeq do SearchByRank($B.botDeqs$, i) instead
88: **return** *null* ▷ dequeue needs a Block discarded by GC

89: IndexTopDeq(Node v, int b, int i) : $\langle$int, int$\rangle$ ▷ returns $\langle b', i'\rangle$ s.t. i^{th} TopDequeue in Block b of v
90: ▷ is i'^{th} TopDequeue of root Block b' or $\langle 0, 0\rangle$ if GC discarded Block needed to find answer
91: **if** $v = root$ **then return** $\langle b, i\rangle$
92: $dir \leftarrow (v.parent.left = v \ ?\ left : right)$
93: $blocks \leftarrow v.rbts.blocks$
94: $blocks_p \leftarrow v.parent.rbts.blocks$
95: ▷ N.B. if any Block required on lines 96–99 is not found, stop and return $\langle 0, 0\rangle$ instead
96: $B_p \leftarrow$ min Block in $blocks_p$ with $end_{dir} \geq b$ ▷ B_p contains TopDequeue
97: $B'_p \leftarrow$ max Block in $blocks_p$ with $end_{dir} < b$
98: $i' \leftarrow i + blocks[b-1].sum_{topDeqs} - blocks[B'_p.end_{dir}].sum_{topDeqs}$
99: **if** $dir = right$ **then** $i' \leftarrow i' + v.parent.left.rbts.blocks[B_p.end_{left}].sum_{topDeqs}$
$- v.parent.left.rbts.blocks[B'_p.end_{left}].sum_{topDeqs}$
100: **return** IndexTopDeq($v.parent$, $B_p.index$, i')

end of $rbts_{old}.state$ to obtain *newstate* and storing an RBT of the TopDequeue responses in $B_{new}.topDeqs$ (line 47). If the split point ($num_{topDeqs}$ in this case) is 0 or greater than the size of the RBT being split, Split returns the original tree and an empty tree. A batch of concurrent BotDequeues is then performed in a similar way at line 48. The batches of items enqueued by operations in B_{new} are contained in *newTopEnqs* and *newBotEnqs*, and they are added to *newstate* by joining them at the left and right of *newstate*, respectively (lines 49–50). Finally, Refresh attempts to CAS $rbts_{new}$ into v as in the non-root case.

CreateBlock creates a new Block B_{new} to be inserted into node v after Block B_{prev}. B_{new} will represent all operations in v's children that are not included in v's Blocks up to B_{prev}. CreateBlock calculates the end_{left} and end_{right} fields of B_{new} by finding the indices of the latest Blocks in v's children (line 58). The sum fields of B_{new} are calculated by adding the sum fields of these end Blocks in v's children (lines 59–62). If B_{new} has no new operations, that is, if the *end* fields of B_{new} and B_{prev} are the same, CreateBlock returns *null* at line 63.

GetTopEnqs retrieves the items of TopEnqueues to be propagated with a new Block into a node v. It first calculates the number of items to be split from the *topEnqs* trees of v's children (lines 71–72), splits those trees to get *rightChunk* and *leftChunk* (lines 74–75), and returns the RBT that results from joining the two chunks together (line 76).

To retrieve its response, a TopDequeue calls **CompleteTopDeq** on the dequeue's leaf Block after the dequeue has propagated to the root (line 22). Because operations help propagate one another, CompleteTopDeq first calls **IndexTopDeq** (line 80), a modified version of N&R's IndexDequeue, which uses recursion to "retrace" the TopDequeue's path from its location in a leaf Block to its location in the root's *blocks*. At each node v along the path up the tree, IndexTopDeq finds the Block B_p in $v.parent$ that contains the TopDequeue at location $\langle b, i\rangle$ in v (where b is a Block index and i is the rank of the TopDequeue in that Block) by searching for the minimum block in $v.parent$'s Blocks whose *end* field is at least b (line 96). The rank i' of the TopDequeue in B_p is found by adding to i the number of TopDequeues in B_p that came from Blocks before B in v (line 98). In the case that v is a right child, line 99 also adds to i' the number of all TopDequeues in B_p that came from v's left sibling, because they will be linearized before B_p's TopDequeues that came from v (see Section 3.3). If any Block that is needed to find the location of the TopDequeue has been discarded by garbage collection, IndexTopDeq returns $\langle 0, 0\rangle$ (line 90). Before discarding the Block, the garbage collection will ensure that some helper writes the TopDequeue's response in the *response* field of the TopDequeue's leaf Block. In this case, CompleteTopDeq returns *null*, indicating the TopDequeue should look there for its response (see line 23).

After IndexTopDeq finds that the TopDequeue is the ith TopDequeue in $root.rbts.blocks[b]$, CompleteTopDeq checks at line 85 if the deque was empty when that operation was performed, and if so, returns *null*. Otherwise, CompleteTopDeq returns the response to the ith TopDequeue in B by searching $B.topDeqs$ for the item with the correct rank (line 86). (We use opposite orderings of the *topDeqs* and *botDeqs* trees, as described in line 87, because both trees hold a chunk of items split off the deque ordered from the bottom to top of the deque, so BotDequeues should take them in this order and TopDequeues should take them in the reverse order; this allows us to use a more uniform ordering of operations when we define the linearization in the next section.)

3.3 Linearization

We shall show that operations within a Block are all concurrent, which allows us to choose how to linearize them. Here, we define the order that is consistent with our pseudocode. We define $D_{top}(B)$, $D_{bot}(B)$, $E_{top}(B)$, and $E_{bot}(B)$ to be the sequences of TopDequeues, BotDequeues, TopEnqueues and BotEnqueues, respectively, in a Block B. We define them recursively after first defining the *direct subblocks* of a Block.

▶ **Definition 1.** *The direct subblocks of a Block $B_i = v.rbts.blocks[i]$ (where $i > 0$) in an internal node v are the blocks that were inserted in $v.left$ with indices from $v.blocks[i-1].end_{left}+1$ to $v.blocks[i].end_{left}$ and the blocks that were inserted in $v.right$ with indices from $v.rbts.blocks[i-1].end_{right}+1$ to $v.rbts.blocks[i].end_{right}$.*

If B is in a leaf node, $D_{top}(B)$ is either a single TopDequeue if a TopDequeue created B, or empty otherwise. If B is in an internal node v, where $B_1^L \cdots B_l^L$ are the direct subblocks of B in $v.left$ and $B_1^R \cdots B_r^R$ are the direct subblocks of B in $v.right$,

$$D_{top}(B) = D_{top}(B_1^L) \cdots D_{top}(B_l^L) \cdot D_{top}(B_1^R) \cdots D_{top}(B_r^R). \tag{3.1}$$

$E_{top}(B)$ is defined similarly except that if B is a leaf Block, $E_{top}(B)$ is either a single TopEnqueue(e) if a call to TopEnqueue(e) created B or the empty sequence otherwise. $D_{bot}(B)$ and $E_{bot}(B)$ are defined similarly to $D_{top}(B)$ and $E_{top}(B)$.

Operations are linearized in the order their Blocks reach the root. Within each Block, Refresh does them in the following order: TopDequeues, BotDequeues, TopEnqueues, and BotEnqueues (lines 47–50). Thus, if $B_0, ..., B_n$ are the blocks that get inserted into the root, ordered by their indices, we define the linearization order as:

$$L_{total} = L(B_0) \cdot L(B_1) \cdots L(B_n), \text{ where } L(B) = D_{top}(B) \cdot D_{bot}(B) \cdot E_{top}(B) \cdot E_{bot}^{rev}(B). \tag{3.2}$$

It is convenient to linearize BotEnqueues in the reverse order of E_{bot}, denoted E_{bot}^{rev}, for uniformity in the pseudocode.

3.4 Garbage Collection

To guarantee the efficiency of our deque, we bound the size of the RBTs at each node so that operations on them can be done efficiently. To do so, we use a more refined version of the garbage collection method of the N&R queue. Specifically, **GC** discards obsolete elements of the RBTs of a node v every $G_v = p_v^2 \lceil \log\ p \rceil$ operations added to v, where p_v is the number of leaves in the sub-tree rooted at v.

Before adding a new Block to v at line 8, 19 or 37, GC is called on v at line 7, 18 or 36. GC first creates an *rbts* object $rbts_{new}$ with fields copied from $v.rbts$. The test at line 106 checks if the new Block to be added to v includes an operation whose rank among operations propagated to v is a multiple of G_v. (Since the number of operations in the new Block is at most p_v and $p_v < G_v$, the test detects when the total number of operations propagated surpasses another multiple of G_v.) If so, lines 107–120 perform garbage on collection $rbts_{new}$.

If v is not the root, lines 112–117 of GC discard unneeded items from the enqueue trees of $rbts_{new}$ by splitting off from their left sides all enqueued items that have already been propagated to v's parent. Since enqueued items are propagated along with the Blocks that represent the enqueues, we can discard the items enqueued by operations in all blocks up to B_{last}, computed on lines 108–110 to be the last Block of v propagated to v's parent. Thus, the number of items to discard is calculated on lines 112–113 by subtracting the number of enqueued items that have already been discarded by previous garbage collection phases from the total number of enqueues in v's Blocks up to B_{last}.

Unlike enqueued items in v, which can be discarded once they are propagated into $v.parent$, a Block of v must be retained if a pending dequeue will need to examine the Block while retracing its path to the root in IndexTopDeq. Discarding a Block in v could prevent a dequeue in the sub-tree rooted at $v.parent$ (or v if $v = root$) from retrieving its response. To avoid this problem, GC first calls **SplitBlock** (line 118), which recursively finds the latest Block B in v that has been propagated to the root by following *end* fields from the last Block

Algorithm 3 Implementation of deque: garbage collection routines.

```
101: GC(Node v, rbts rbts_old, Block B_new, Block B_prev) : rbts
102:     ▷ garbage collect the RBTs of node v before B_new gets appended to v.blocks
103:     rbts_new ← new rbts object with fields copied from rbts_old
104:     sum_prev ← B_prev.sum_topEnqs + B_prev.sum_botEnqs + B_prev.sum_topDeqs + B_prev.sum_botDeqs
105:     sum_new ← B_new.sum_topEnqs + B_new.sum_botEnqs + B_new.sum_topDeqs + B_new.sum_botDeqs
106:     if sum_prev mod G_v ≥ sum_new mod G_v then ▷ trigger garbage collection
107:         if v ≠ root then                          ▷ discard propagated enqueues
108:             B_p ← MaxBlock(v.parent.rbts.blocks)
109:             dir ← (v.parent.left = v ? left : right)
110:             B_last ← rbts_new.blocks[B_p.end_dir]
111:             if B_last ≠ null then                 ▷ skip if GC already done
112:                 num_oldTopEnq ← B_last.sum_topEnqs − rbts_new.discarded_topEnqs
113:                 num_oldBotEnq ← B_last.sum_botEnqs − rbts_new.discarded_botEnqs
114:                 rbts_new.discarded_topEnqs ← B_last.sum_topEnqs
115:                 rbts_new.discarded_botEnqs ← B_last.sum_botEnqs
116:                 ⟨∗, rbts_new.topEnqs⟩ ← Split(rbts_new.topEnqs, num_oldTopEnq)
117:                 ⟨∗, rbts_new.botEnqs⟩ ← Split(rbts_new.botEnqs, num_oldBotEnqs)
118:         i ← SplitBlock(v).index                   ▷ find index of oldest Block to keep
119:         Help(v)                                   ▷ help pending dequeues in v's subtree
120:         ⟨∗, rbts_new.blocks⟩ ← SplitByIndex(blocks, i) ▷ discard blocks with indices < i
121:     return rbts_new

122: SplitBlock(Node v) : Block                        ▷ returns v's latest Block propagated to root
123:     if v = root then B ← MaxBlock(root.rbts.blocks)
124:     else
125:         B_p ← SplitBlock(v.parent)
126:         B ← (v.parent.left = v ? v.rbts.blocks[B_p.end_left] : v.rbts.blocks[B_p.end_right])
127:     return (B = null ? MinBlock(v.rbts.blocks) : B)

128: Help(Node v) ▷ complete pending dequeues in v.parent's sub-tree
129:     for each leaf l in subtree rooted at v.parent (or at v if v = root) do
130:         B_cur ← MaxBlock(l.rbts.blocks)
131:         if B_cur.index ≠ 0 then                   ▷ do not help dummy Block
132:             B_prev ← l.rbts.blocks[B_cur.index − 1]
133:             response ← null
134:             if B_cur.sum_topDeqs ≠ B_prev.sum_topDeqs and Propagated(l, B_cur.index) then
135:                 response ← CompleteTopDeq(l, B_cur.index)
136:             else if B_cur.sum_botDeqs ≠ B_prev.sum_botDeqs and Propagated(l, B_cur.index) then
137:                 response ← CompleteBotDeq(l, B_cur.index)
138:             if response ≠ null then
139:                 B_cur.response ← response

140: Propagated(Node v, int b) : boolean ▷ has v's Block with index b been propagated to root?
141:     if v = root then return true
142:     else
143:         blocks_p ← v.parent.rbts.blocks
144:         dir ← (v.parent.left = v ? left : right)
145:         if MaxBlock(blocks_p).end_dir < b then return false
146:         else                                      ▷ Block b has been propagated to parent
147:             B_p ← min Block in blocks_p with end_dir ≥ b ▷ B_p exists, by line 145's test
148:             return Propagated(v.parent, B_p.index)
```

in the root. Then, GC calls Help, which helps pending dequeues in all leaves in the subtree rooted at v's parent to compute their responses. Finally, GC splits $rbts_{new}.blocks$ to discard all Blocks strictly older than B (line 120).

Help ensures that the response to any pending dequeue that may need Blocks that are being discarded is written in the dequeue's leaf Block. Help only has to help dequeues that have been propagated to the root, since only those might need the discarded Blocks. For each leaf l in the subtree of $v.parent$ (or v if $v = root$) (see line 129), **Help** checks at line 134 or 136 if l contains a dequeue (using the first condition) that has been propagated to the root (using the call to **Propagated**). If so, Help completes the dequeue by calling CompleteTopDeq or CompleteBotDeq at line 135 or 137, and records the response in the *response* field of the dequeue's leaf Block (line 139). If CompleteTopDeq or CompleteBotDeq returns *null*, due to a missing Block on the path used by IndexTopDeq (or IndexBotDeq) at line 80, Help does nothing further since another process must have already helped the dequeue.

Help uses **Propagated** to check if a leaf's dequeue operation has been propagated to the root. Propagated recurses from the leaf to the root, finding in each node along that path the Block that contains the operation, returning false if there is no such Block.

4 Correctness

The goal of the correctness proof is to show that each operation is propagated to the root and applied to the state of the deque before the operation terminates. This allows us to argue that the linearization ordering of Equation (3.2) is valid. Garbage collection requires us to prove that no operation needs to access any of the discarded information. Due to space constraints, some details of the correctness proof are deferred to the full version.

4.1 Basic Properties

We first prove some basic properties of nodes' *rbts* fields. The first lemma describes how the *blocks* RBT is updated, ensuring the RBT's Blocks always have consecutive indices.

▶ **Lemma 2.** *If a node's blocks field is updated from T to T' and I is the set of indices in the RBT T, then the set of indices in T' is $\{I \cap [i, \infty)\} \cup \{\max(I) + 1\}$ for some $i \geq 0$ and the only new Block is the one with index $\max(I) + 1$.*

Proof Sketch. Each node's *blocks* field initially contains one dummy Block with $index = 0$. The *blocks* field of a node can change only when the *rbts* field of the node is updated at line 10, 20 or 51. It is straightforward to check that each such change modifies the RBT by optionally splitting off some blocks with indices below some threshold i (if GC is called) and adding one new Block with index $\max(I) + 1$. ◀

▶ **Definition 3.** *Define $v.blocks[i]$ to mean the Block with index i that was in $v.rbts.blocks$ at some point during the execution. By Lemma 2, this Block is unique.*

Next, we show that end_{left} and end_{right} fields of Blocks in a node are in sorted order. This ensures that Definition 1, which defines direct subblocks of a Block, makes sense.

▶ **Lemma 4.** *If a Block with index $h > 0$ has been added to an internal node v's blocks then $v.blocks[h].end_{left} \geq v.blocks[h-1].end_{left}$ and $v.blocks[h].end_{right} \geq v.blocks[h-1].end_{right}$.*

Proof. Let B_{h-1} and B_h be the Blocks with indices $h-1$ and h installed in v by two calls to Refresh, R_{h-1} and R_h, respectively. Since R_h performs a successful CAS at line 51, it must have read $v.rbts$ at line 28 *after* R_{h-1} performed its successful CAS at line 51. Thus, R_h read

$v.left.rbts$ at line 29 after R_{h-1} did. By Lemma 2, the maximum index in $v.left.rbts.blocks$ can only increase over time. Thus, the value stored in $B.end_{left}$ at line 58 of R_{h-1}'s call to CreateBlock is less than or equal to the value stored in $B'.end_{left}$ by R_h's call to CreateBlock, as required. The argument for end_{right} is identical. ◀

▶ **Definition 5.** *The* subblocks of a Block B *are defined recursively to be either direct subblocks of B (as defined by Definition 1), or subblocks of the direct subblocks of B.*

▶ **Definition 6.** *A Block B is* propagated *to node v if it is a subblock of some Block that has been inserted into $v.rbts.blocks$.*

▶ **Definition 7.** *A Block B* contains *an operation if the Block inserted by the operation into a leaf is a subblock of B.*

Next, we show the *sum* fields of a Block B in node v correctly capture the number of operations contained in v's Blocks up to and including Block B. We use the notation $E_{top}(blocks[i...j])$ for $E_{top}(blocks[i]) \cdots E_{top}(blocks[j])$, and likewise for E_{bot}, D_{top}, and D_{bot}.

▶ **Invariant 8.** *If v is a node in the ordering tree and $B = v.rbts.blocks[i]$, then*

$$B.sum_{topEnqs} = |E_{top}(v.blocks[0...i])|, \qquad B.sum_{botEnqs} = |E_{bot}(v.blocks[0...i])|,$$
$$B.sum_{topDeqs} = |D_{top}(v.blocks[0...i])|, \text{and} \qquad B.sum_{botDeqs} = |D_{bot}(v.blocks[0...i])|.$$

Proof Sketch. The invariant holds initially, since each *blocks* RBT contains a single empty Block with no operations and *sum* fields set to 0. Lemma 2 ensures each update of the RBT adds only a single Block B. We check that each such step preserves the invariant. When B is added to a leaf's *blocks* RBT at line 10 or 20, it contains a single operation, which is recorded in the appropriate *sum* field at lines 4–6 or 15–17. If an internal node's *blocks* RBT is updated at line 51, the new Block B was created by the call to CreateBlock on line 34 and inserted into the RBT at line 37. By Definition 1 and Lemma 2, the direct subblocks of $v.blocks[0 \dots B.index]$ are $v.left.blocks[0 \dots B.end_{left}]$ and $v.right.blocks[0 \dots B.end_{right}]$. It follows that B's *sum* fields are computed correctly at lines 59–62 of the CreateBlock. ◀

The following result is easy to prove by induction on the height of the node(s) containing B and B', using Definition 1 and Lemma 4.

▶ **Lemma 9.** *The sets of subblocks of two Blocks B and B' at the same level of the ordering tree are disjoint.*

Since the set of operations contained in a Block B is the set of all operations that appear in subblocks of B in the leaves of the ordering tree, we have the following corollary.

▶ **Corollary 10.** *For any node v in the ordering tree, if $i \neq j$, $v.blocks[i]$ and $v.blocks[j]$ cannot contain the same operation.*

4.2 Operations are Propagated to the Root

In this section, we prove that operations are correctly propagated to the root of the *ordering tree*, where they are applied to the *state* field that represents the sequence of items in the deque. This requires showing (in Invariant 12) that enqueued items are added to a node's *topEnqs* or *botEnqs* RBT at the same time the corresponding Enqueues are added to the node's *blocks* RBT. But first, we must show that garbage collection does not discard any required information. Since Blocks in a node are propagated upwards in order by their index and also discarded by GC in order by their index, the following invariant implies that the GC routine only discards Blocks (at line 120) that are already propagated to the root.

▶ **Invariant 11.** *For any non-root node v, the minimum Block in $v.rbts.blocks$ and any Block returned by* SplitBlock(v) *are subblocks of a Block that has been inserted into the root.*

Proof Sketch. Initially, each node has a single dummy Block whose *index*, end_{left} and end_{right} fields are 0. Thus, every Block is a subblock of the dummy Block in the root.

We show that each step maintains the invariant. The minimum Block in v is only modified because GC discards Blocks (at line 120) whose indices are smaller than the result of the call to SplitBlock at line 118. Assuming the invariant held for the response of the SplitBlock, the new minimum Block in v also satisfies the invariant. Proving the claim for the response of a SplitBlock is an easy induction on the depth of v, since SplitBlock simply returns a subblock of the Block returned by SplitBlock$(v.parent)$. ◀

The next invariant says that the items stored in the *topEnqs* tree of a non-root node v are the arguments of all TopEnqueues that have propagated to v, except that some of the oldest ones may have been discarded by garbage collection. The $discarded_{topEnqs}$ field keeps track of how many have been discarded. The second claim of the invariant ensures that we never discard items from the *topEnqs* tree before they have been propagated to v's parent. The invariant also holds if all occurrences of *top* are replaced by *bot* (and the proof is identical).

▶ **Invariant 12.** *Let v be a non-root node and let k be the maximum index of any Block in $v.rbts.blocks$. The in-order traversal of $v.rbts.topEnqs$ yields the arguments of some suffix of the sequence $E_{top}(v.blocks[1 \dots k])$, where the suffix is obtained by removing the first $v.rbts.discarded_{topEnqs}$ elements of the sequence. Moreover, $v.rbts.discarded_{topEnqs}$ is less than or equal to the $sum_{topEnqs}$ field of the last propagated Block in v.*

Proof Sketch. Initially, $v.rbts.blocks$ has one Block with *index* 0, $v.rbts.topEnqs$ is empty, and $v.rbts.discarded_{topEnqs}$ is 0. We show each update to $v.rbts$ preserves the invariant.

If v is a leaf, the argument is straightforward because the *rbts* field is updated by an Enqueue or Dequeue that adds one Block containing a single operation at line 8 or 19, and *topEnqs* gets one new item at line 9 in the case of TopEnqueue or none in the case of other operations.

If v is an internal node, we must show that a Refresh operation's successful CAS on line 51 preserves the invariant. The CAS adds a new Block B_{new} created by the call to CreateBlock at line 34 to v's *blocks* RBT and simultaneously joins *newTopEnqs* to the right side of v's *topEnqs* RBT (see line 42). We must therefore prove the following claim.

▷ Claim 12.1. The call to GetTopEnqs on line 39 returns a RBT *newTopEnqs* whose in-order traversal yields the arguments of the sequence $E_{top}(B_{new})$.

This claim is proved by tracing the code of GetTopEnqs and using the fact that the *sum* fields are accurate (by Invariant 8). The proof also uses the fact that the required enqueued items have not been discarded from v's children's *topEnqs* trees, which follows from the induction hypothesis that the second claim of the invariant holds prior to updating v's *rbts* field. ◀

When a Block reaches the root, instead of appending its enqueued items to *topEnqs* or *botEnqs*, the Refresh attaches them to the left and right ends of the *state* RBT. Similarly, it detaches the appropriate number of items for dequeues and saves them in the *topDeqs* and *botDeqs* fields of the Block (lines 45–50). This allows us to prove the folllowing key invariant that the *state* field represents the state an abstract deque would have after sequentially performing, in their linearization order, all the operations that have been propagated to the

root so far. Moreover, the *topDeqs* and *botDeqs* fields of each Block contain the non-empty results of all the dequeue operations contained in that Block, a fact that is crucial for showing that Dequeues return results consistent with the linearization. Recall that $L(B)$ is defined in Equation (3.2) to be the linearization order of operations in a Block B of the root.

▶ **Invariant 13.** *Let $B_0, \ldots, B_n$ be all the Blocks that have so far been added to the root's blocks RBT. The in-order traversal of root.rbts.state is the state of an initially empty deque after the sequential execution $L(B_0) \cdot L(B_1) \cdots L(B_n)$. Moreover, for $0 \leq i \leq n$, the reverse in-order traversal of B_i.topDeqs gives the sequence of non-empty responses to the TopDequeues of $L(B_i)$ in this sequential execution, and the in-order traversal of B_i.botDeqs gives the sequence of non-empty responses to the BotDequeues of $L(B_i)$ in this sequential execution.*

Proof Sketch. The claim holds initially: the root has a single Block B_0 with no operations and the *state* is empty. We show each addition of a Block to the root preserves the invariant.

Consider the Refresh whose CAS on line 51 adds B_n to the root's *blocks* tree. This CAS simultaneously updates the *state* field to $rbts_{new}.state$, which was constructed by performing two Split and two Join operations on the previous state (see lines 45–50).

By Invariant 8, line 45 sets $num_{topDeqs}$ to $|D_{top}(B_n)|$. Thus, line 47 splits $|D_{top}(B_n)|$ elements off the right end of the old *state* and stores these values, which are the non-empty responses to operations the TopDequeues of B_n, into $B_n.topDeqs$. Line 48 handles the BotDequeues of $D_{bot}(B_n)$ similarly. By Claim 12.1, line 49 adds the items enqueued by the TopEnqueues of $E_{top}(B_n)$ to the right end of the state. Line 50 handles the BotEnqueues similarly. The order in which the operations are applied matches $L(B_n)$ defined in Equation (3.2). ◀

A double Refresh on a node v guarantees that all operations that had previously propagated to v's children are propagated to v: if both Refreshes fail their CAS, then a concurrent Refresh must have successfully propagated the operations. The following two lemmas formalize this argument, similarly to previous papers that use a double Refresh [1, 18, 26].

▶ **Lemma 14.** *All operations contained in Blocks of v's children when a Refresh(v) performed line 28 are contained in Blocks of v when the Refresh returns true at line 35 or performs a successful CAS at line 51.*

Proof. Suppose an operation *op* is contained in a Block B of v's left child v' when Refresh performs line 28. (The proof for the right child of v is identical.) Refresh gets a snapshot of $v'.rbts$ at line 29. Then, the call to CreateBlock on line 34 finds the Block B_L in this snapshot of $v'.rbts.blocks$ with the maximum index at line 54. By Lemma 2, $B'.index \geq B.index$. At line 58, CreateBlock writes $B'.index$ into the end_{left} field of the new Block B_{new}.

If the Refresh returns true at line 35, then $B_{prev}.end_{left} = B_{new}.end_{left} \geq B.index$. If the Refresh performs a successful CAS at line 51 then that CAS successfully installs Block B_{new} in v's Blocks tree. Either way, v contains a Block whose end_{left} field is greater than or equal to $B.index$. It follows from Definition 1 and Lemma 4, that B is a direct subblock of some Block that has been added to v. Thus, *op* is contained in a Block of v, as required. ◀

▶ **Lemma 15.** *All operations contained in Blocks of v's children when a Propagate(v) was invoked are contained in Blocks of v when the Propagate completes line 25.*

Proof. If either call to Refresh on line 25 returns true, then the claim follows from Lemma 14. So, suppose the Propagate's calls R_1 and R_2 to Refresh both return false, which can happen at line 31 or 51. In either case, some other Refresh must have changed $v.rbts$ between line 28

of the Refresh and the time it returns false. Let R'_1 and R'_2 be instances of Refresh that update $v.rbts$ during R_1 and R_2, respectively. Because R'_2 performs a successful CAS, it must have read $v.rbts$ at line 28 after R'_1 modifies it (and hence after R_1 executes line 28). Moreover, R'_2 performs its successful CAS before R_2's failed CAS. Thus, the claim follows from Lemma 14 applied to R'_2. ◀

The following lemma implies that each operation's linearization point is within the interval of time when the operation is executing. It follows easily from Lemma 15.

▶ **Lemma 16.** *Propagate ensures that the operation that has called it is contained in a Block in the root before the Propagate terminates.*

It follows from Invariant 13 and Lemma 16 that each operation is performed on the root's *state* when a Block containing the operation reaches the root and the root's *state* is updated.

4.3 Retrieval of a Dequeue's Response and Linearizability

In this section, we show each dequeue returns the response it would in the sequential execution L_{total}. We first show IndexTopDeq correctly locates the TopDequeue in the root.

▶ **Lemma 17.** *If $v.rbts.blocks[b]$ contains at least i TopDequeues and it is a subblock of a root Block, then IndexTopDeq(v, b, i) returns $\langle b', i' \rangle$ such that the ith TopDequeue in $D_{top}(v.rbts.blocks[b])$ is the i'th TopDequeue in $D_{top}(root.rbts.blocks[b'])$, or $\langle 0, 0 \rangle$ in the case that any of the Blocks looked up at lines 96–99 have been discarded.*

Proof Sketch. We use induction on the depth of v. If v is the root, line 91 satisfies the claim. Suppose the claim holds for $v.parent$ and Blocks looked up at lines 96–99 are found. By Definition 1, $v.rbts.blocks[b]$ is a direct subblock of the Block B_p found on line 96. It follows from Invariant 8 that line 98 computes i' to be the rank of the ith TopDequeue within $D_{top}(v.rbts.blocks[b])$ plus the number of TopDequeues in the subblocks of B_p in v that precede $D_{top}(v.rbts.blocks[b])$. By Definition 1, the i'th TopDequeue in $D_{top}(B_p)$ is the ith TopDequeue in $D_{top}(v.rbts.blocks[b])$ if v is the left child of $v.parent$. If v is the right child of $v.parent$, the i'th dequeue in $D_{top}(B_p)$ is shifted at line 99 by the number of TopDequeues in the left subblocks of B_p. Thus, the ith TopDequeue in $D_{top}(v.rbts.blocks[b])$ is the i'th TopDequeue in $D_{top}(B_p)$. By the induction hypothesis, the call to IndexTopDeq$(v.parent, B_p.index, i')$, at line 100 returns the required pair. ◀

The next two lemmas show that each dequeue returns a result consistent with the linearization, either because the dequeue reads the result directly from the *botDeqs* or *topDeqs* field of the root Block that contains the dequeue, or because some other operation has retrieved the response from there and stored it in the dequeue's leaf Block as part of the helping performed in GC. The argument about GC's helping must also carefully show that information is never discarded before all operations that need the information have been helped.

▶ **Lemma 18.** *A call to CompleteTopDeq$(leaf, h)$ returns either the response the TopDequeue in $leaf.rbts.blocks[h]$ would receive in the sequential execution L_{total} or null. Moreover, it returns null only if one of the Blocks looked up in lines 96–99 or line 83 have been discarded. The same claim holds for CompleteBotDeq.*

Proof Sketch. Before calling CompleteTopDeq at line 22 or 135, the call to Propagate at line 21 or the test at line 134 ensures the TopDequeue to complete is already propagated to the root. By Lemma 17, the call to IndexTopDeq at line 80 returns the TopDequeue's location in

the root, or $\langle 0, 0\rangle$ if some Block needed has been discarded. In the latter case, or if the search at line 83 does not find the required Block, CompleteTopDeq returns *null* at line 88, and the lemma is satisfied. Otherwise, line 83 finds the root Block B containing the TopDequeue. Then, if $B.topDeqs$ contains fewer than i elements (line 85), CompleteTopDeq returns *empty*, since there are fewer than i non-empty TopDequeues in B, by Invariant 13. Otherwise, Invariant 13 ensures the response of the ith TopDequeue in B is the $(n - i + 1)$th element in $B.topDeqs$, which is returned by line 86. The argument for BotDequeues is similar. ◀

▶ **Lemma 19.** *If a call to CompleteTopDeq on a leaf Block returns null, then either the TopDequeue in that leaf Block has terminated or the value returned by that TopDequeue in the sequential execution L_{total} has been written into the TopDequeue's leaf Block. The same claim holds for CompleteBotDeq.*

Proof Sketch. We use induction on the number of completed calls to CompleteTopDeq. (The proof for CompleteBotDeq is identical.) Consider a call C to CompleteTopDeq and assume the claim holds for all calls that complete before C does. By Lemma 18, C can return *null* only if one of the Blocks it searches for on line 83 or lines 96–99 has been discarded.

First, suppose C fails to find a required Block B in some node v on line 83, 96 or 97. B is either the Block that contains the TopDequeue or the preceding Block. B must have been discarded by some call to GC(v) whose call to SplitBlock(v) on line 118 returned a Block B' with a larger index than B. By Invariant 11, B' was already propagated to the root, so B must also have been propagated to the root. If the TopDequeue has not terminated, the call to Help(v) on line 119 performs CompleteTopDeq on that TopDequeue. If that call of CompleteTopDeq returned *null*, the claim holds by the induction hypothesis. Otherwise, it returned the correct response of the TopDequeue by Lemma 18 and this response was recorded in the *response* field of the dequeue's leaf Block, so the claim holds.

The case where C returns *null* because a required Block is not found in the left sibling of v on line 99 of IndexTopDeq can be argued similarly. (This is where we use the fact that Help helps all processes in the subtree of $v.parent$, not just in v's subtree.) ◀

Finally, we prove the main result of our correctness proof.

▶ **Theorem 20.** *The deque is linearizable.*

Proof. By Corollary 10 applied to the root, L_{total} is a permutation of operations in the concurrent execution. Lemma 16 ensures that the permutation includes all completed operations, since they are propagated to the root before they terminate. Lemma 16 also ensures that an operation appears in L_{total} before any operations that begin after it terminates. Finally, we show each TopDequeue returns the same response as it would in the sequential execution L_{total}. (The proof for BotDequeues is identical.) This follows from Lemma 19 if the call to CompleteTopDeq at line 22 returns *null*, or from Lemma 18 otherwise. ◀

5 Space Complexity

We bound the amount of memory accessible through the ordering tree data structure. We assume that the number of operations performed on the deque can be represented in binary using $O(1)$ memory words. We first bound the number of operations and Blocks in a node v's *blocks* tree. We say that GC on v *succeeds* if the test at line 106 is satisfied and the operation that called GC successfully installs the resulting RBTs tuple in v. A successful GC discards at line 120 any Blocks that have propagated to the root. Thus, each Block that remains after a successful GC has an operation that was in the process of propagating that Block to the root

at the time the $RBTs$ field of v was read prior to GC. Since each process has at most one pending operation, at most p_v Blocks remain immediately after GC, where p_v is the number of leaves in the subtree rooted at v. Since each Block contains at most p_v operations, there are at most p_v^2 operations remaining immediately after GC. The next GC on v succeeds after another $G_v = p_v^2\lceil \log\ p\rceil$ operations propagate to v. Thus, the number of operations in v's *blocks* tree can never exceed $p_v^2 + G_v = O(p_v^2 \log p)$. The test on line 63 ensures that each Block added to a node has at least one subblock, and it follows that each Block contains at least one operation. Thus, the number of Blocks in v is also $O(p_v^2 \log p)$ at all times.

When GC on a non-root node v succeeds, all items in v's enqueue RBTs that have been propagated to v's parent are discarded at lines 116–117. So, following the GC, at most p_v enqueued items remain (at most one per process). Since GC succeeds every $p_v^2\lceil \log p\rceil$ operations added to v (line 106), there are at most $p_v + p_v^2\lceil \log p\rceil$ items in v's enqueue RBTs at any time. Thus, the total size of $v.RBTs$ is $O(p_v^2 \log p)$ for each non-root node v.

Each Block in the root contains a RBT of responses to dequeues in that Block. Since it was shown above that the *blocks* RBT of the root contains $O(p^2 \log p)$ operations, the total size of the dequeue trees in all of the root's Blocks is $O(p^2 \log p)$. The space used by the *state* tree in the root is $O(q_{max})$ where q_{max} is the maximum size of the deque. The root's total space, including its *state*, *blocks*, and its Blocks' dequeue trees, is $O(q_{max} + p^2 \log\ p)$.

The value of p_v of a non-root node v at each level of our *ordering tree*, from top to bottom, is $\frac{p}{2}, \frac{p}{4}, ..., 1$, and there are $2, 4, ..., p$ non-root nodes at each level of the tree. Since each non-root node v uses $O(p_v^2 \log p)$ space, the total space used by the non-root nodes is $O(\sum_{i=1}^{\lceil logp\rceil} 2^i \left(\frac{p}{2^i}\right)^2 \log p) = O(p^2 \log p \sum_{i=1}^{\lceil logp\rceil} \frac{1}{2^i}) = O(p^2 \log p)$. Together with the space used by the root, the total space usage of the ordering tree is $O(q_{max} + p^2 \log p)$.[1]

6 Step Complexity

RBT operations on the root's *state* tree take $O(\log q_{max})$ steps. Since the sizes of all other RBTs are polynomial in p, operations on them each take $O(\log p)$ steps.

We first bound the step complexity of operations *excluding garbage collection*. Each operation performs $O(\log p)$ steps to insert the operation at a leaf of the ordering tree and then calls Propagate. Propagate calls Refresh at most twice at each of $O(\log p)$ nodes along a path from the leaf to the root. At non-root nodes, each Refresh, including the calls to the subroutines CreateBlock, GetTopEnqs and GetBotEnqs, does $O(1)$ RBT operations and $O(1)$ other steps, for a total of $O(\log p)$ steps. At the root, Refresh also does RBT operations on the *state* tree (lines 47–50), so it takes $O(\log p + \log q_{max})$ steps. Thus, Propagate takes $O(\log^2 p + \log q_{max})$ steps in total. A TopDequeue also calls CompleteTopDeq at line 22, which calls IndexTopDeq. IndexTopDeq searches a *blocks* tree at each level of the ordering tree, for a total of $O(\log^2 p)$ steps and the rest of CompleteTopDeq searches two RBTs in $O(\log p)$ steps. Summing up, each enqueue or dequeue takes $O(\log^2 p + \log q_{max})$ steps, excluding calls to GC.

Now we consider the contribution of GC to the amortized step complexity of operations. First, we bound the steps taken by routines called by GC. Propagated takes at most $O(\log^2 p)$ steps to search *blocks* RBTs (lines 145–147) at each level of the ordering tree. In the worst

[1] In addition, processes could have pointers to additional objects in local memory. For example, in a pathological execution where processes fall asleep during a Refresh holding pointers in their local memory to old, totally disjoint states of the deque, this could add an additional $\Theta(pq_{max})$ memory usage.

case, Help(v) performs a *blocks* search (line 130), calls Propagated (line 134 or 136) and calls CompleteTopDeq (line 135) or CompleteBotDeq (line 137) for the $2p_v$ leaf descendants of $v.parent$. So Help(v) takes $O(p_v \log^2 p)$ steps. SplitBlock(v) takes $O(\log^2 p)$ steps since it recurses to the root from v, searching a *blocks* RBT at each level.

If garbage collection is not triggered at line 106, GC takes $O(1)$ steps. If it is triggered, GC performs $O(\log p)$ steps on RBTs (lines 108–117) and calls SplitBlock (line 118), Help (line 119), and SplitByIndex on a *blocks* tree (line 120) for a total of $O(p_v \log^2 p)$ steps. Each of the p_v processes whose leaves are descendants of v satisfy the trigger at line 106 only once every G_v operations added to v, and each of them performs $O(p_v \log^2 p)$ steps in that case. So, all processes perform a total of $O(p_v^2 \log^2 p)$ steps doing GC at v once every G_v operations propagated to v. So the amortized number of steps for GC at node v is $O(p_v^2 \log^2 p / G_v) = O(\log p)$ per operation. Each operation does GC at each level of the tree, the total amortized number of steps spent on GC is $O(\log^2 p)$ per operation. The total amortized step complexity of a deque operation, including GC, is thus $O(\log^2 p + \log q_{max})$.

Our deque is wait-free because each recursion recurses from a leaf of the ordering tree to the *root* or vice versa, and its only loop (line 129) iterates through a finite set. Also, all RBT operations are applied to a local snapshot of the RBT, and are therefore wait-free.

7 Elimination

As an optimization, it would be straightforward to incorporate elimination [14] into our deque. If any block (in any non-root node) contains both enqueues and dequeues on the same end of the deque, they can be eliminated and there is no need to propagate them further up the tree. For example, suppose a Refresh builds a block B_{new} that represents 5 TopDequeues and 8 TopEnqueues. We could eliminate 5 pairs of operations by reducing the $sum_{topDeqs}$ field of the block by 5, removing 5 elements from the $newTopEnqs$ tree constructed on line 39 using Split, and storing those 5 elements in a new $topDeqs$ RBT field of the block, as is done in the Blocks of the root. Thus, only the 3 remaining TopEnqueues would be propagated further. Elimination also requires some changes to the CompleteTopDeq function: when tracing a TopDequeue up the tree, it would detect if the operation got eliminated at some node, and then use the operation's rank among TopDequeues eliminated in that Block to select the response from the Block's $topDeqs$ RBT. The eliminated operations would not appear in the E_{top} and D_{top} sequences of the Block that are used to define the linearization ordering L_{total}. The eliminated operations would be linearized when the Block that eliminates them is installed in the node, instead of when they are propagated to the root.

8 Open Questions

It would be interesting to experimentally compare the performance of the new deque to existing lock-free deques. Such experiments could also measure the effects of elimination on throughput. Our new deque is designed to ensure time bounds even in worst-case executions. However, this comes at the cost of operations performing more steps in the best case (for example, when an operation runs with no contention). Could the deque be made adaptive so that its step complexity depends on the number of concurrent operations, rather than on the number of processes in the system? This might be achieved by having operations choose a leaf at which to inject an operation (as in [1]) rather than having a statically assigned leaf for each process. (This might also handle the case where the set of processes is not known in advance.) Or perhaps the fast-path slow-path methodology of [21] could be used to make the

best case faster while maintaining good bounds in the worst case. An interesting theoretical direction would be narrowing the gap between the $\Omega(\log p)$ lower bound on the amortized step complexity of operations [19] and our $O(\log^2 p + \log q)$ upper bound. Can we design (implicit or explicit) representations of batches of operations for other data structures so that ordering trees yield sublinear-time operations for still more data structures?

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 Ole Agesen, David Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. DCAS-based concurrent deques. *Theory of Computing Systems*, 35(3):349–386, 2002. doi:10.1007/S00224-002-1058-2.

3 Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001. doi:10.1007/S00224-001-0004-Z.

4 Hagit Attiya and Arie Fouren. Lower bounds on the amortized time complexity of shared objects. In *Proc. 21st International Conference on Principles of Distributed Systems*, volume 95 of *LIPIcs*, pages 16:1–16:18, 2017. doi:10.4230/LIPICS.OPODIS.2017.16.

5 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 17.1. MIT Press, fourth edition, 2022.

6 David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent deques. In *Proc. 14th International Conference on Distributed Computing*, volume 1914 of *LNCS*, pages 59–73, 2000. doi:10.1007/3-540-40026-5_4.

7 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, pages 233–246, 2015. doi:10.1145/2676726.2676963.

8 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.

9 Matthew Graichen, Joseph Izraelevitz, and Michael L. Scott. An unbounded nonblocking double-ended queue. In *Proc. 45th International Conference on Parallel Processing*, pages 217–226, 2016. doi:10.1109/ICPP.2016.32.

10 Michael Barry Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999. Available from http://i.stanford.edu/TR/CS-TR-99-1624.html.

11 Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978. doi:10.1109/SFCS.1978.3.

12 Andreas Haas. *Fast Concurrent Data Structures Through Timestamping*. PhD thesis, University of Salzburg, 2015.

13 Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18(3):189–207, 2006. doi:10.1007/S00446-005-0144-5.

14 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *Journal of Parallel and Distributed Computing*, 70(1):1–12, 2010. doi:10.1016/J.JPDC.2009.08.011.

15 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

16 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003. doi:10.1109/ICDCS.2003.1203503.

17 Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998. doi:10.1145/277697.277735.

18 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.

19 Siddhartha V. Jayanti, Robert E. Tarjan, and Enric Boix-Adserà. Randomized concurrent set union and generalized wake-up. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 187–196, 2019. doi:10.1145/3293611.3331593.

20 Donald E. Knuth. *The Art of Computer Programming*, chapter 2.2.1. Addison-Wesley, third edition, 1997.

21 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proc. 17th ACM Symposium on Principles and Practice of Parallel Programming*, pages 141–150, 2012. doi:10.1145/2145816.2145835.

22 Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. Lincheck: A practical framework for testing concurrent data structures on JVM. In *Proc. 35th International Conference on Computer Aided Verification, Part I*, volume 13964 of *LNCS*, pages 156–169, 2023. doi:10.1007/978-3-031-37706-8_8.

23 Paul Martin, Mark Moir, and Guy Steele. DCAS-based concurrent deques supporting bulk allocation. Technical Report SMLI TR-2002-11, Sun Microsystems, oct 2002. Available from https://dl.acm.org/doi/10.5555/1698157.

24 Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *Proc. 9th International Euro-Par Conference on Parallel Processing*, volume 2790 of *LNCS*, pages 651–660, 2003. doi:10.1007/978-3-540-45209-6_92.

25 Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 103–112, 2013. doi:10.1145/2442516.2442527.

26 Hossein Naderibeni and Eric Ruppert. A wait-free queue with polylogarithmic step complexity. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 124–134, 2023. doi:10.1145/3583668.3594565.

27 Niloufar Shafiei. Non-blocking array-based algorithms for stacks and queues. In *Proc. 10th International Conference on Distributed Computing and Networking*, volume 5408 of *LNCS*, pages 55–66, 2009. doi:10.1007/978-3-540-92295-7_10.

28 Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, 2008. doi:10.1016/J.JPDC.2008.03.001.

29 Robert Endre Tarjan. *Data Structures and Network Algorithms*, chapter 4.2. SIAM, Philadelphia, USA, 1983. doi:10.1137/1.9781611970265.

30 R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

Reliable Broadcast Despite Mobile Byzantine Faults

Silvia Bonomi
Sapienza University of Rome, Italy

Giovanni Farina
Sapienza University of Rome, Italy

Sébastien Tixeuil
Sorbonne Université, CNRS, LIP6, Institut Universitaire de France, Paris, France

Abstract

We investigate the solvability of the Byzantine Reliable Broadcast and Byzantine Broadcast Channel problems in distributed systems affected by Mobile Byzantine Faults. We show that both problems are not solvable even in one of the most constrained system models for mobile Byzantine faults defined so far. By endowing processes with an additional local failure oracle, we provide a solution to the Byzantine Broadcast Channel problem.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Byzantine fault-tolerance, Reliable Broadcast, Mobile Byzantine Faults

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.18

Related Version *Full Version*: `https://hal.science/hal-04277831` [7]

Funding *Giovanni Farina*: This work was partially supported by the *CBFTFDS project – Concrete Byzantine Fault Tolerance and Forecasting in Distributed Systems* (Bando di Ateneo per la Ricerca 2022, Sapienza University of Rome) and partially funded by the EURASIA project by the Italian MAECI.
Sébastien Tixeuil: This work was partially funded by the ANR projects SAPPORO (ref. 2019-CE25-0005-1) and ESTATE (ref. ANR-16-CE25-0009-03).

1 Introduction

Byzantine Reliable Broadcast (BRB) is a fundamental primitive in fault-tolerant distributed systems ensuring that all correct processes eventually deliver the same message from a defined sender regardless of its correctness. Defined by Bracha [12] as a building block for a Byzantine-tolerant consensus protocol, BRB has been widely adopted and investigated since then, thanks to its ability to prevent arbitrarily (i.e., Byzantine) faulty processes from *equivocating* by sending different messages to different processes. It has been introduced as a *one-shot* primitive that allows a pre-defined process in the system to spread a single message and generalized as a Byzantine Broadcast Channel (BBC) primitive [14] to allow every process to spread an arbitrary number of messages. BRB has been used to construct several fault-tolerant distributed solutions, solving more complex problems such as register abstractions, consensus problems, and distributed ledgers. Thus, it has been analyzed in the literature from various perspectives, such as minimizing bandwidth consumption [2], or latency [20, 1].

A fundamental perspective to consider is the investigation of the feasibility of BRB and BBC when assuming no permanent failures. In this paper, we are interested in analyzing BRB and BBC solvability considering a *dynamic process failure model*, i.e., a model in which every process may potentially fail and recover, causing a potentially continuous change in a process's failure state throughout the system's lifetime. Some examples of systems

considering dynamic process failures are crash-recovery systems [28, 5], self-stabilizing systems [15, 16], and Mobile Byzantine tolerant systems [17, 6]. In this work, we consider the *Mobile Byzantine Failure* (MBF) model, in which all processes may alternate between periods of correct behavior and periods of arbitrary behavior (i.e., Byzantine). Indeed, the failure state of processes is governed by an external attacker capable of compromising and controlling a set of processes in the system, and such a set is dynamic. The MBF model captures some of the features of the most frequent attacks targeting distributed systems and related countermeasures, where the process's faults are primarily due to external malicious causes rather than internal misbehavior, and tools such as software rejuvenation techniques [21], intrusion detection systems [23], and trusted execution environments [29] are available.

Despite several fundamental distributed problems have been analyzed in the literature considering the MBF model (i.e., Byzantine agreement [17, 6], approximate Byzantine agreement [32, 9], and registers emulation [8]), to the best of our knowledge the BRB problem has never been studied so far in such settings.

Thus, our objective in this paper is the investigation of BRB and BBC in the presence of MBFs. In particular, our contributions are:

1. we formalize the *Mobile Byzantine Reliable Broadcast* (MBRB) and *Mobile Byzantine Broadcast Channel* (MBBC) as a natural extension of the BRB and BBC specifications to deal with MBFs. Indeed, the standard specifications for BRB and BBC primitives consider a *static failure model*, where every process is either permanently correct or faulty;
2. we prove several impossibility results, mainly showing that MBRB and MBBC cannot be implemented without additional knowledge provided by a powerful oracle reporting about processes' failure state;
3. we introduce such a powerful oracle and provide a protocol for solving MBBC in a synchronous round-based system;
4. we analyze a weaker MBBC specification that can be realized without the oracle.

Let us note that being a natural extension of BRB and BBC primitives, the MBRB and MBBC primitives prevent faulty processes from equivocating, namely from sending different information to different processes, and can be used as building block for other fault-tolerant primitives. For example, MBRB/MBBC primitives can extend mobile Byzantine fault-tolerant register abstractions to support Byzantine clients [8]. Our work not only offers an analysis of a specific problem but also provides several insights for other distributed system problems where the failure state of a process is dynamic and partially or entirely unknown. We consider relatively strong assumptions in our system model, the same as those considered in related work, in order determine fundamental solvability conditions. Relaxation of most of these assumptions has already been partially investigated [11].

The rest of the paper is structured as follows. After reviewing related work on implementations of the BRB primitive and contributions considering mobile Byzantine failures in Section 2, we formalize the system model in Section 3. We introduce the new specifications for the Mobile Byzantine Reliable Broadcast and the Mobile Byzantine Broadcast Channel problems in Section 4. Section 5 presents some impossibilities for the specifications we defined. To overcome some of the identified impossibilities and solve the Mobile Byzantine Broadcast Channel problem, we consider a powerful oracle, we propose a protocol in Section 6, and we analyze a weaker Mobile Byzantine Broadcast Channel specification that is realizable without any oracle in Section 7. Due to space constraints, some of the proofs are delegated to the companion technical report [7].

2 Related Work

The Byzantine Reliable Broadcast (BRB) abstraction has been introduced by Bracha [12] as a building block for a Byzantine-tolerant consensus protocol in a distributed system where at most f processes are permanently arbitrary (Byzantine) faulty. Thanks to its ability to guarantee agreement among correct processes over the set of delivered messages, a BRB primitive has been used as a building block from several fault-tolerant solutions, and has been intensely investigated under several system and failure models, with the final aim of extending its power and optimizing different performance metrics.
Imbs and Raynal [20] proposed a protocol that improves latency (in terms of the number of rounds of message exchanges) compared to Bracha. Guerraoui et al. [19] relaxed the BRB specification, allowing each property to be violated with a fixed and arbitrarily small probability. Backes and Cachin [3] and Raynal [26] discussed extensions of the BRB problem; the former assuming both Byzantine faulty processes and fail-stop failures, the latter distinguishing between two different kinds of Byzantine behaviors, i.e. those attempting to prevent the liveness and those attempting to prevent the safety of the BRB. Recently, Guerraoui et al. [18] and Li et al. [22] extended BRB to distributed systems with dynamic membership: in any given view (i.e. set of participating processes, governed by the processes themselves), the set of Byzantine processes remains the same; however, two consecutive views allow for different sets of Byzantine processes. By contrast, our work considers a static system membership (i.e., a fixed set of processes participating in the protocol) but a dynamic failure model, where Byzantine processes may change (that is, recover, and get Byzantine again) during the *same* view. To the best of our knowledge, all existing BRB protocols that assumed arbitrary process failures, except the aforementioned works by Guerraoui et al. [18] and Li et al. [22], considered a *static failure model* i.e., they assumed that the set of Byzantine processes does not change.

Mobile Byzantine Failure (MBF) models have been introduced to capture various types of faults, such as external attacks, virus infections, or even arbitrary behaviors caused by software bugs, using a single model encompassing detection and rejuvenation capabilities. In all these models, failures are abstracted by an omniscient adversary that can control up to f mobile Byzantine agents. Every agent is located in a process and makes it Byzantine faulty until the omniscient adversary decides to move it to another process. The main differences between existing MBF models are in the power of the omniscient adversary (i.e., when it can move the agents) and in the awareness that every process has about its failure state. Most MBF models considered *round-based computations* and can be classified according to Byzantine mobility constraints: under *constrained mobility* [13] the adversary can move agents only when protocol messages are sent (similarly to how viruses would propagate), while under *unconstrained mobility* [4, 6, 17, 24, 31, 27] agents do not move with messages but rather during specific phases of the round. More in detail, Reischuk [27] considered malicious agents stationary for a given period; Ostrovsky and Yung [24] introduced the notion of mobile viruses and defined the adversary as an entity that can inject and distribute faults; finally, Garay [17], Banu et al. [4], Sasaki et al. [31], and Bonnet et al. [6] considered that processes execute synchronous rounds and mobile agents can move from one process to another in a specific phase of the round, which subsequently affects each process's ability to adhere to the algorithm. As a result, the set of Byzantine faulty processes at any given moment is limited in size; however, its composition may change from one round to the next, and the impact of past compromises may linger if not properly addressed by the protocol. The aforementioned works [17, 4, 31, 6] also differ due to the assumption about

the knowledge that processes have about their previous infection. In the Garay model [17], a process can detect its infection after the agent leaves it. Conversely, Sasaki et al. [31] investigated a model where processes cannot detect when agents leave. Finally, Bonnet et al. [6] considered an intermediate setting where not faulty processes control the messages they send (in particular, they send the same message to all destinations, and they do not send spurious information). Bonomi et al. [10, 11] decoupled algorithm rounds from Mobile Byzantine agent movement (*round-free model*). The problems analyzed under MBF models are Byzantine agreement [17, 4, 31, 6], approximate Byzantine agreement [32, 30, 9], and Byzantine-tolerant registers [10, 8, 11]. To the best of our knowledge, no efforts have been made to investigate the BRB problem in the presence of MBFs. All existing works that assume MBFs rely on some kind of best-effort communication subsystem (i.e., no guarantees exist when a process is controlled by a Mobile Byzantine agent), potential equivocations and omissions introduced by faulty processes are directly addressed by the main investigated primitive (e.g., consensus, register). The existence of a BRB primitive can simplify the definition of other mobile Byzantine fault-tolerant primitives, similar to the case of the static failure model [12].

3 System Model

We consider a distributed system composed of a set of n processes $\Pi = \{p_1, p_2 \dots p_n\}$, each associated with a unique identifier.

Processes communicate through message passing. We assume that a process can communicate with any other process through a *reliable*, *authenticated*, *point-to-point link* abstraction [14]. This means that messages sent over such channels cannot be altered, dropped, or duplicated, and the identity of the sender cannot be forged. A reliable authenticated point-to-point link abstraction exposes two operations: *(i)* $\mathsf{P2P.send}(p_{rcv}, m)$ which sends the message m to the receiver process p_{rcv}, and *(ii)* $\mathsf{P2P.deliver}(p_{snd}, m)$ which notifies the reception of the message m from a sender process p_{snd}.

We measure the time according to a fictional global clock $\mathbb{T}$ (not accessible to processes) spanning over the set of natural numbers $\mathbb{N}$. We refer to the starting time of the system as t_0, the i-th time instant since the beginning of the execution as t_i, and a period of time between time t_b and t_e as $T_{b,e} := [t_b, t_e) : t_b, t_e \in \mathbb{T}; t_b < t_e$.

Each process executes a distributed protocol $\mathcal{P}$ consisting of a set of local algorithms. Each algorithm in $\mathcal{P}$ is represented by a finite state automaton whose transitions correspond to computation and communication steps. A computation step denotes a computation executed locally by a given process, while a communication step denotes the sending or receiving of a message. Computation steps and communication steps are generally called *events*. Each process maintains a set of variables. This set and the current value of those variables denote the *state* of a process.

▶ **Definition 1** (Local Execution History). *A local execution history is an alternating sequence $s_0, e_0, s_1, e_1, \dots$ of states and events of a process p_i, such that state s_{j+1} results from state s_j by executing event e_j.*

We assume that the local algorithms composing $\mathcal{P}$ are stored in a tamper-proof read-only memory.

Processes may fail and we assume that they are affected by *Mobile Byzantine Failures* (MBF). That is, we assume the existence of an omniscient adversary that controls up to $f > 0$ mobile Byzantine agents and that can "move" such agents from one set of processes to

another. When the adversary places a Byzantine agent on a process p_i, the agent takes control of p_i, letting it behave arbitrarily. For example, p_i may omit to send/receive messages, alter the content of messages, alter its process state regardless of its local algorithm, and execute arbitrary code. However, we assume that the mobile Byzantine agents cannot compromise the code stored in the tamper-proof memory. Thus, when the Byzantine agent leaves p_i, p_i resumes executing its local algorithm correctly (albeit from a possibly corrupted state). We assume that the adversary can move each mobile agent independently of the others. Still, any agent must remain on a process for a period of time lasting at least $\Delta_s \in \mathbb{Q}^+$ (rational positive numbers), i.e., once arrived, an agent compromises a node for at least Δ_s consecutive time units, and when $\Delta_s < 1$ we have that an agent can move multiple times in the same time unit. As an example, if $\Delta_s = 2$ we have that every mobile Byzantine agent must remain on the same process for at least 2 consecutive time units, while $\Delta_s = \frac{1}{2}$ means that the agent may move $\left\lceil \frac{1}{\Delta_s} \right\rceil = 2$ times in a time unit and compromise $\left\lceil \frac{1}{\Delta_s} \right\rceil = 2$ different processes in the same time unit.
Let us note that, in the MBF model, no single process is guaranteed to remain correct forever and we may have processes that alternate between correct and incorrect behavior infinitely often. This fundamental difference from the classical static Byzantine failure model commands to redefine the notion of correct and faulty processes (i.e., *the process failure states*).

▶ **Definition 2** (Faulty process). *A process p_i is said to be* faulty *at time t_k if it is controlled by a mobile Byzantine agent at time t_k. By extension, if at each time between t_b and t_e, process p_i is* faulty*, then p_i is faulty during the period $T_{b,e}$.*

When a process p_i is faulty, it may execute a protocol $\mathcal{P}' \neq \mathcal{P}$, and its local state may be altered arbitrarily.

We denote by $B(t)$ the set of faulty processes at time t and by $B(T_{b,e})$ the set of faulty processes during the whole period $T_{b,e}$ (i.e., $B(T_{b,e}) = \bigcap_i B(t_i)$ for $b \leq i < e$).

▶ **Definition 3** (Correct process). *A process p_i is* correct *when it is not faulty, that is, p_i is correct at time t_k if it is not controlled by a Byzantine agent at time t_k. Similarly, a process p_i is correct in the period $T_{b,e}$ if it remains correct between times t_b and t_e.*

Let us remark that when a process p_i is correct, it executes $\mathcal{P}$ but potentially it may start its execution from a compromised state (due to a previous corruption performed by a mobile Byzantine agent).
We denote by $C(t_k)$ the set of correct processes at time t_k and by $C(T_{b,e})$ the set of correct processes throughout the period $T_{b,e}$ (that is, $C(T_{b,e}) = \bigcap_i C(t_i)$ for $b \leq i < e$).

Note that, due to the mobility of Byzantine agents, every process may potentially alternate between correct and faulty states infinitely often. To this aim, we also introduce the notion of *infinitely often correct processes*:

▶ **Definition 4** (Δ_c-Infinitely often correct process). *Let $\Delta_c \in \mathbb{N}^+$. A process p_i is* Δ_c-infinitely often correct *if, for every time t_j, there exists a following period $T_{b,e}$ lasting at least Δ_c where p_i is correct. Formally: $\forall t_j \in \mathbb{T},\ \exists t_b, t_e$ such that $t_b > t_j,\ t_e - t_b \geq \Delta_c,\ p_i \in C(T_{b,e})$.*

Informally, the notion of Δ_c-infinitely often correct process captures the possibility that a process is not permanently faulty, but correct for at least Δ_c units of time after mobile Byzantine agents have left it.

In the following, we will consider several alternative settings for our system model:

- **system timing assumptions**: we consider either a *synchronous* (SYNC) or an *asynchronous* (ASYNC) system. When considering a synchronous system, we assume that there is an upper bound on the time required to perform local computation on the processes and an upper bound on the time required by a message to be delivered via a P2P link, both of them known by all processes. In addition, we assume that the computation evolves in sequential synchronous rounds $r_1, r_2, \ldots, r_j, \ldots$. Every round r_j is divided into three phases: *(i) send* where processes transmit messages to their intended receivers, *(ii) receive* where processes collect messages sent during the send phase of the current round, and *(iii) compute* where processes process received messages, and prepare those that need to be sent in the following round. Contrarily, in an asynchronous setting, we are not assuming any upper bound, and the computation progresses as soon as an event is generated by a process.
- **mobile Byzantine agent synchronization**: we consider three different types of mobility with different degrees of synchronization between mobile Byzantine agents. In particular, we will consider movement that are either *synchronized* (S-MOB$^+$), *synchronous* (S-MOB), or *asynchronous* (A-MOB) that abstract MBF models existing in the literature. In the A-MOB model, mobile Byzantine agents move independently and once the movement occurs, the agent remains at the destination node for at least Δ_s, with Δ_s unknown to the processes (see ITU model in [10]). In the S-MOB model, mobile Byzantine agents move independently, and, also in this case, once the movement happens the agent remains on the destination node for at least Δ_s. Unlike the previous case, Δ_s is known to the processes (see the ITB model in [10]). The S-MOB$^+$ model is a particular case of the S-MOB model specific for synchronous systems where the computation evolves in synchronous rounds. Indeed, in this case Δ_s is expressed in terms of round, and mobile Byzantine agents can move only between two consecutive rounds, i.e. after the computation phase of a round r_i and before the send phase of round r_{i+1}[1](see Garay's MBF model [17]). Let us stress that in the S-MOB$^+$ setting every process is either faulty or correct for an entire round. Therefore, for ease of presentation, we say that a process is *faulty or correct in the round* r_k in the S-MOB$^+$ systems and extend the notation of $C(t)$ and $B(t)$ accordingly, that is, with $C(r_k)$ and $B(r_k)$, respectively, referring to the sets of correct and faulty processes in the round r_k. Furthermore, we measure the time with the number of rounds.
- **failure awareness**: we assume that every process p_i is either *aware* or *unaware* about a mobile Byzantine agent moving away from p_i. We abstract this knowledge by introducing two different local oracles that reveal information to process p_i. Specifically, we consider: *basic failure awareness* ($\mathcal{O}_{BFA}$) and *full failure awareness* ($\mathcal{O}_{FFA}$). In the $\mathcal{O}_{BFA}$ case, a process p_i knows when (i.e., in which time unit) a mobile agent moves away from p_i; in the $\mathcal{O}_{FFA}$ case, a processes p_i additionally know when the agent arrived to p_i (i.e., p_i know the entire period $T_{b,e}$ in which it was faulty).

More formally:

▶ **Definition 5** (*Basic Failure Awareness Oracle* $\mathcal{O}_{BFA}$)**.** *If a mobile Byzantine agent leaves from a process* p_i *at time* t_j*, then the failure awareness oracle* $\mathcal{O}_{BFA}$ *generates a* CURED() *event on* p_i *at time* t_{j+1}*.*

Observe that $\mathcal{O}_{BFA}$ informs p_i as soon as p_i becomes free from mobile Byzantine agents, and thus allows p_i to take corrective actions (*e.g.* to avoid spreading compromised information).

[1] The agents' movements are thus synchronized with the synchronous rounds.

However, $\mathcal{O}_{BFA}$ does not provide any information about the length of the period p_i was faulty.

▶ **Definition 6** (*Full Failure Awareness Oracle* $\mathcal{O}_{FFA}$). *If a mobile Byzantine agent takes control of a process p_i at time t_j and leaves p_i at time t_k, then the full failure awareness oracle $\mathcal{O}_{FFA}$ generates a* CURED() *event on p_i at time t_{k+1}, and returns the time label t_j when invoking operation* FAULTY_AT().

For the sake of notation, we refer to setting where no oracle is available as $\mathcal{O}_{NFA}$. Let us remark that both $\mathcal{O}_{BFA}$ and $\mathcal{O}_{FFA}$ are *local* oracles, i.e., they provide information to the actual process where the events occurred; thus, a process p_i is not aware of the failure state of any other process p_j.

Note that the assumptions considered in our system model are equivalent to or less constrained than those in other works dealing with mobile Byzantine agents [17, 4, 31, 6]. The only exceptions are the $\mathcal{O}_{FFA}$ oracle and the notion of Δ_c-*infinitely often correct* process, which have not been considered before.

In the remainder of the paper, we will characterize the specific setting considered in terms of system timing assumptions, agent synchronization, and failure awareness by specifying a triple $\langle \alpha, \beta, \gamma \rangle$ where $\alpha \in \{\texttt{SYNC}, \texttt{ASYNC}\}$, $\beta \in \{\texttt{A-MOB}, \texttt{S-MOB}, \texttt{S-MOB}^+\}$ and $\gamma \in \{\mathcal{O}_{BFA}, \mathcal{O}_{FFA}, \mathcal{O}_{NFA}\}$. With slight abuse of notation, we will use " $*$ " in a triple when the specific dimension is not relevant to prove our claims.

4 Mobile BRB and BBC Specification

Informally *Byzantine Reliable Broadcast* (BRB) [12, 14] is a communication primitive that enables all processes of a distributed system to agree on the delivery of a single message disseminated by a pre-defined process called the *source*, while the *Byzantine Broadcast Channel* (BBC) [14] primitive extends BRB allowing all processes to disseminate an arbitrary number of messages so that all correct processes eventually deliver the same set of messages [2].

Let us note that in the original BRB and BBC specifications the source is either always correct or always faulty in a given execution. Conversely, in our settings, it is possible that the source of a message changes its failure state multiple times (even during a single broadcast instance) making the original specification no more suitable. Thus, we extend the BRB and BBC, by formalizing the *Mobile Byzantine Reliable Broadcast* (MBRB) and the *Mobile Byzantine Broadcast Channel* (MBBC) problems to capture challenges imposed by mobile Byzantine faults. We aim to specify two communication primitives accessible by every process and exposing the MBRB/MBBC.BROADCAST(m) and MBRB/MBBC.DELIVER(s,m) operations, where m is a message and s is a process identifier. We say that a process p_i "MBRB/MBBC-broadcasts a message m" when it executes MBRB/MBBC.BROADCAST(m), and p_i "MBRB/MBBC-delivers a message m from p_s" when p_i generates the MBRB/MBBC.DELIVER(s, m) event. Similarly to other communication primitives, the MBRB/MBBC-BROADCAST operation is triggered to disseminate a message, while MBRB/MBBC-DELIVER notifies message deliveries. We associate two additional parameters to both primitives, $\Delta_b \in \mathbb{N}^+$ and $\Delta_c \in \mathbb{N}^+$, characterizing the length of two periods (detailed in the specifications' properties). We use the character "*" in our specifications when the actual value of the reference parameter is irrelevant.

[2] The formal specification of BRB and BBC primitives are provided in the Appendix A.

Informally, a MBRB(Δ_b, Δ_c) communication primitive guarantees that, given a source process p_s and a message m generated by p_s while it is correct (for at least Δ_b time units), m is reliably delivered by any Δ_c-infinitely often correct process p_j in a period where p_j is correct. Similarly to BRB, this primitive is specified by considering an instance for every message generated by the identified source. More formally, a MBRB(Δ_b, Δ_c) communication primitive must guarantee the following properties:

- (Δ_b, Δ_c)-*Validity*: If there exists a period $T_{i,j}$ lasting at least Δ_b where a process p_s is correct in $T_{i,j}$ and executes MBRB.BROADCAST(m), then at least one Δ_c-infinitely often correct process p_d eventually executes MBRB.DELIVER(s,m) while correct.
- *No duplication*: Every process p_d executes MBRB.DELIVER(s,$*$) at most once when correct, namely p_d MBRB-delivers at most one message from p_s among all times $t_k \in \mathbb{T}$ such that $p_d \in C(T_{k,k+1})$.
- Δ_b-*Integrity*: If a process p_d is correct at time t_k and executes MBRB.DELIVER(s,m), then either p_s was correct in $T_{i,j} = [t_i, t_{i+\Delta_b})$, with $t_i \leq t_k$, and executed MBRB.BROADCAST(m) at time t_i, or p_s was faulty at some $t_i \leq t_k$.
- *Consistency*: If some process is correct at time t_k and executes MBRB.DELIVER(s, m), and another process is correct at time t_l and executes MBRB.DELIVER(s, m'), then $m = m'$.
- Δ_c-*Totality*: If some process is correct at time t_k and executes MBRB.DELIVER($s, *$), then every Δ_c-infinitely often correct process eventually executes MBRB.DELIVER($s, *$).

The MBBC communication primitive is the natural extension of the BBC and its specification extends the one of the MBRB. In particular, the MBBC primitive guarantees that multiple messages generated by a source process (while it is correct for at least Δ_b consecutive time units) will be eventually delivered by any process p_j that is Δ_c-infinitely often correct in a period in which p_j is correct. More formally, a MBBC(Δ_b,Δ_c) communication primitive must guarantee the following properties:

- (Δ_b, Δ_c)-*Validity*: If there exists a period $T_{i,j}$ lasting at least Δ_b where a process p_s is correct in $T_{i,j}$ and executes MBRB.BROADCAST(m), then at least one Δ_c-infinitely often correct process p_d eventually executes MBRB.DELIVER(s,m) while correct.
- *No duplication*: Every process p_d executes MBBC.DELIVER(s,m), with message m and source s, at most once when correct, namely, it MBBC-delivers a message m from p_s at most once among all times t_k such that $p_d \in C(T_{k,k+1})$.
- Δ_b-*Integrity*: If a process p_d is correct at time t_k and executes MBRB.DELIVER(s,m), then either p_s was correct in $T_{i,j} = [t_i, t_{i+\Delta_b})$, with $t_i \leq t_k$, and executed MBRB.BROADCAST(m) at time t_i, or p_s was faulty at some $t_i \leq t_k$.
- Δ_c-*Agreement*: If some process is correct at time t_k and executes MBRB.DELIVER(s, m), then every Δ_c-infinitely often correct process eventually executes MBRB.DELIVER(s, m).

Note that the specifications rule the MBRB/MBBC.DELIVER(s, m) operations in times when processes are correct. Operations executed when a process is faulty cannot be controlled and thus are not relevant to the specification. Furthermore, note that when a process is controlled by a mobile Byzantine agent, it may execute arbitrary code and alter its local memory. Such a process has no information about what occurred when compromised (except the fact of being previously compromised in case an oracle is available). This makes the implementation of the presented communication primitives particularly challenging and will lead to proving several impossibility results that are specific to mobile Byzantine faults in the following sections.

5 Impossibility Results

This section presents several impossibility results for the MBRB and MBBC problems. In particular, Theorems 7 and 9 prove the impossibility of solving both MBRB and MBBC if the system is asynchronous, or if the agents' movements are asynchronous. Then, assuming a synchronous system and synchronized agents, Theorems 10 and 12 state the impossibility of solving MBRB with the strongest failure oracle we considered, $\mathcal{O}_{FFA}$, and the impossibility of solving MBBC with the weaker failure oracle, $\mathcal{O}_{BFA}$. These latter impossibilities arise from the fact that a correct process cannot infer other processes' failure state from their behavior. Thus, they cannot distinguish messages that must be delivered from those that can be safely dropped. Table 1 provides an overview of the impossibilities proved in this Section based on the specific considered settings.

▶ **Theorem 7.** *There exists no protocol $\mathcal{P}$ implementing the Mobile Byzantine Reliable Broadcast (resp. Mobile Byzantine Broadcast Channel) in $\langle$ASYNC, S-MOB, $\mathcal{O}_{FFA}\rangle$.*

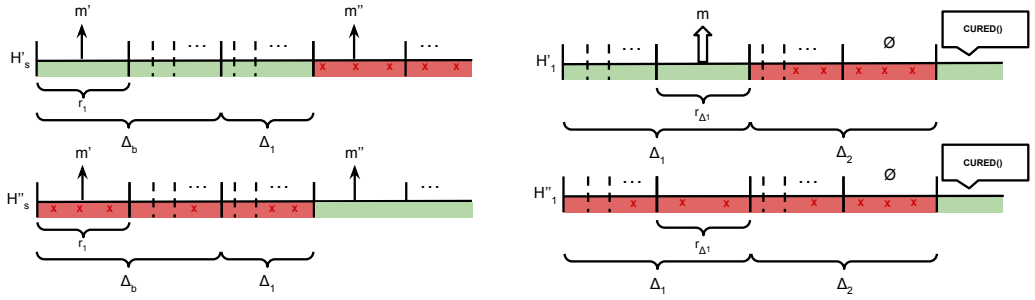
Let us note that Theorem 7 holds assuming the most constrained agent's mobility model available in an asynchronous system (i.e., S-MOB) and the most powerful failure oracle ($\mathcal{O}_{FFA}$) considered. It follows that the MBRB and MBBC problems cannot be solved in ASYNC assuming a less constrained environment, as stated in the following Corollary.

▶ **Corollary 8.** *There exists no protocol $\mathcal{P}$ implementing the Mobile Byzantine Reliable Broadcast (resp. Mobile Byzantine Broadcast Channel) in $\langle$ASYNC, $M, O\rangle$, with $M \in \{$A-MOB, S-MOB$\}$ and $O \in \{\mathcal{O}_{FFA}, \mathcal{O}_{BFA}\}$.*

▶ **Theorem 9.** *There exists no protocol $\mathcal{P}$ implementing the Mobile Byzantine Reliable Broadcast (resp. Mobile Byzantine Broadcast Channel) in $\langle$SYNC, A-MOB, $\mathcal{O}_{FFA}\rangle$.*

▶ **Theorem 10.** *If $\Delta_b \in \mathbb{N}^+$ and $\Delta_b \geq 2$ rounds, then there exists no protocol $\mathcal{P}$ implementing a Mobile Byzantine Reliable Broadcast primitive in $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{FFA}\rangle$.*

Proof. For the sake of contradiction, let us assume that such a protocol $\mathcal{P}$ exists. Let us consider the local execution history $\mathcal{H}'_s$ of a process p_s that is *correct* for $\Delta_b \geq 2$ rounds and executes MBRB.Broadcast(m') in round r_1. Subsequently, p_s remains correct for the successive Δ_1 rounds, it gets permanently *faulty* from round $r_{\Delta_b+\Delta_1+1}$ (namely $\forall r_j \in [r_{\Delta_b+\Delta_1+1}, \infty)$, $p_s \in B(r_j)$), and it executes MBRB.Broadcast(m'') in round $r_{\Delta_b+\Delta_1+1}$. We remark that the failure state of any process may change unexpectedly due to the movement of a Byzantine agent. Let us consider another local execution history $\mathcal{H}''_s$ of process p_s where the failure state of p_s evolves in the opposite way from $\mathcal{H}'_s$, that is process p_s is *faulty* in rounds $r_j \in [r_1, r_{\Delta_b+\Delta_1}]$ and executes MBRB.Broadcast(m') in round r_1; subsequently, p_s is permanently *correct* from round $r_{\Delta_b+\Delta_1+1}$ (namely $\forall r_j \in [r_{\Delta_b+\Delta_1+1}, \infty), p_s \in C(r_j)$) and executes MBRB.Broadcast(m'') in round $r_{\Delta_b+\Delta_1+1}$. Notice that in both histories p_s executes the MBRB.Broadcast operation only once while correct. We provide a graphical representation of the two histories in Figure 1a. Let us consider a process $p_1 \neq p_s$ that is correct for the entire lifetime of the system (i.e. $\forall r_j$, $p_1 \in C(r_j)$), thus p_1 is also an Δ_c-infinitely often correct process for any value of $\Delta_c \in \mathbb{N}$. The two execution histories $\mathcal{H}'_s$ and $\mathcal{H}''_s$ are indistinguishable to p_1 because the same operations and events occurred on p_s. Process p_1 is not aware of the failure state of p_s (i.e. it has no access to the failure oracle on p_s). Even defining an algorithm $\mathcal{A}$ that allows process p_s to share the information obtained from $\mathcal{O}_{FFA}$ with process p_1 through the point-to-point primitive, process p_1 cannot distinguish an execution of $\mathcal{A}$ where p_s is correct and reveals a previous faulty state, from another where p_s is faulty, and maliciously reports the same information.



(a) Graphical representations for Theorem 10. **(b)** Graphical representations for Theorem 12.

Figure 1 Graphical representations for Theorems' proof.

According to the *Validity* property of the MBRB specification, process p_1 executing $\mathcal{P}$ must MBRB-deliver a message from p_s considering both histories because process p_s MBRB-broadcasts a message when correct. If $\mathcal{P}$ makes process p_1 eventually MBRB-deliver message m', then the *Validity* property is violated in $\mathcal{H}''_s$, because process p_1 never MBRB-delivers m'' (according to the *No-duplication* property) that is broadcast when p_s is correct. If $\mathcal{P}$ makes process p_1 eventually MBRB-deliver message m'', then the *Validity* property is violated in $\mathcal{H}'_s$ for the same reason. This is a contradiction and the claim follows regardless of the value of Δ_b and Δ_c. ◀

Theorem 10 states the impossibility in solving MBRB assuming the most constrained assumptions we considered. Corollary 11 extends the result to less constrained settings.

▶ **Corollary 11.** *If $\Delta_b \in \mathbb{N}^+$ and $\Delta_b \geq 2$ rounds, then there exists no protocol $\mathcal{P}$ implementing a Mobile Byzantine Reliable Broadcast primitive in $\langle$`SYNC`, `S-MOB`$^+, \mathcal{O}_{BFA}\rangle$ or in $\langle$`SYNC`, `S-MOB`$, *\rangle$.*

▶ **Theorem 12.** *If $\Delta_b \in \mathbb{N}^+$ and $\Delta_b \geq 2$ rounds, then there exists no protocol $\mathcal{P}$ implementing a Mobile Byzantine Reliable Channel primitive in $\langle$`SYNC`, `S-MOB`$^+, \mathcal{O}_{BFA}\rangle$.*

Proof. For the sake of contradiction, let us assume that such a protocol $\mathcal{P}$ exists. Let us assume a permanently correct process p_s (i.e. $\forall r_j,\ p_s \in C(r_j)$) that executes MBBC.Broadcast(m) in rounds r_1. Let us consider the local execution history $\mathcal{H}'_1$ of a process p_1 that is *correct* in rounds $r_j \in [r_1, r_{\Delta_1}]$, $\Delta_1 \in \mathbb{N}$, and executes MBBC.Deliver(m) in round r_{Δ_1}; subsequently, p_1 gets *faulty* for Δ_2 consecutive rounds, $\Delta_2 \in \mathbb{N}$, it wipes its local state (i.e. initialises all the process variables) in round $r_{\Delta_1+\Delta_2}$, and it gets *permanently correct* from round $r_{\Delta_1+\Delta_2+1}$ (namely $\forall r_i \in [r_{\Delta_1+\Delta_2+1}, \infty),\ p_1 \in C(r_i)$).
Let us consider another local execution history $\mathcal{H}''_1$ of process p_1 that is *faulty* in rounds $r_j \in [r_1, r_{\Delta_1+\Delta_2}]$ and it wipes its local state in round $r_{\Delta_1+\Delta_2}$; subsequently, p_1 gets *permanently correct* from round $r_{\Delta_1+\Delta_2+1}$ (namely $\forall r_j \in [r_{\Delta_1+\Delta_2+1}, \infty),\ p_1 \in C(r_j)$). We provide a graphical representation in Figure 1b. In round $r_{\Delta_1+\Delta_2+1}$, process p_1 has the same local state in both histories and the $\mathcal{O}_{BFA}$ oracle generates the same CURED() event on process p_1. Process p_1 does not know what happened during the previous rounds. It is even defining an algorithm $\mathcal{A}$ that allows any process p_i to share and retrieve the state and events occurred on the process through the point-to-point primitive: process p_i can execute such a protocol either as correct or as faulty, and the two executions would be indistinguishable by any other process.

Table 1 Summary of the solvability results.

(a) MBRB.

	ASYNC	SYNC	
S-MOB$^+$		$\mathcal{O}_{BFA}$	$\mathcal{O}_{FFA}$
		✗ (Cor. 11)	✗ (Th. 10)
S-MOB	✗ (Cor. 8)	$\mathcal{O}_{BFA}$	$\mathcal{O}_{FFA}$
		✗ (Cor. 11)	✗ (Cor. 11)
A-MOB	✗ (Cor. 8)	✗ (Th. 9)	

(b) MBBC.

	ASYNC	SYNC	
S-MOB$^+$		$\mathcal{O}_{BFA}$	$\mathcal{O}_{FFA}$
		✗(* Sec 7) (Th. 12)	✓ **(Th. 16)**
S-MOB	✗ (Cor. 8)	$\mathcal{O}_{BFA}$	$\mathcal{O}_{FFA}$
		✗	?
A-MOB	✗ (Cor. 8)	✗ (Th. 9)	

According to the *Validity* property of the MBBC specification, process p_1 executing $\mathcal{P}$ must MBBC-deliver message m from p_s in both histories. In round $r_{\Delta_1+\Delta_2+1}$ process p_1 has the same local state on both histories, thus it can act in one only way, specifically it can command or not process p_i to deliver message m from p_s. In the positive case, the protocol violates the *No duplication* property in history $\mathcal{H}'_1$, in the negative case the *Validity* property is violated by the protocol in $\mathcal{H}''_1$. This leads to a contradiction and the claim follows regardless to the value of Δ_1,Δ_2, and Δ_c. ◀

Discussion. Contrarily to what we could expect, the MBRB and MBBC problems are impossible to solve in settings (*e.g.*, $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{NFA/BFA}\rangle$) where the register abstraction and consensus problems are solvable [17, 4, 31, 6, 10, 8, 11]. The intuition behind this is that other problems addressed under the MBF model have a semantics that do not require to execute a particular operation (the delivery of a message in our case) at most once and depending on a precedent failure state of the process. Indeed, both the register abstractions and consensus set constraints on a local value stored by the processes (respectively, the shared value and the decided value) but no primitive operation is associated with their update in their specification. Contrarily, MBRB and MBBC introduce constraints on the deliveries of messages that depend on the actual and previous failure states of the processes, generating thus symmetry conditions that are impossible to break without violating one of the properties characterizing the specification. In particular, the main challenge is to ensure that a single broadcast instance does not generate multiple deliveries to the same process while it is correct. Another counter-intuitive result is that considering a setting stronger than the one considered in related works (*e.g.*, $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{FFA}\rangle$), the MBRB problem is impossible to solve while the MBBC one is possible (see Section 6). In the static Byzantine failure model (where every process is always either correct or faulty in a given execution), the channel specification extends the broadcast one allowing multiple broadcast from the same source. As a matter of fact, in the mobile Byzantine failure model such an extension is less constrained with respect to the broadcast: in MBRB, every process can execute only one broadcast operation for the entire lifetime of the system, whereas MBBC allows multiple broadcasts from the same source; if a process is faulty and executes a broadcast, then it is not allowed to execute a subsequent broadcast when correct in the future in the MBRB specification (*No duplication* property), while it is in MBBC. Finally, note that other primitives, such as consensus or register abstractions, are not useful in solving the MBRB/MBBC problems. Consider again the execution depicted in Figure 1a, correct process may agree or may store a set of delivered messages (according to the MBRB/MBBC specifications) but a single process (p_s in the example), in the settings we characterized, cannot infer if it has already delivered or not a message if it was previously compromised.

6 A Protocol for MBBC in $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{FFA}\rangle$

Theorem 12 and Corollary 11 motivate the definition of a stronger local oracle than those considered in related work dealing with mobile Byzantine faults, $\mathcal{O}_{FFA}$: both MBRB and MBBC are impossible to solve in the ($\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{NFA/BFA}\rangle$) settings. Theorem 10 states the impossibility in solving MBRB even in ($\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{FFA}\rangle$). This Section investigates the remaining open problem-setting: the solvability of MBBC in ($\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{FFA}\rangle$). Specifically, we start by defining $\mathcal{P}_{MBBC-RB}$, a protocol implementing the MBBC(Δ_b, Δ_c) communication primitive. Then, we prove its correctness and fault-tolerance optimality.

6.1 $\mathcal{P}_{MBBC-RB}$: Protocol Description

$\mathcal{P}_{MBBC-RB}$ is an extension of Bracha's algorithm [12] aimed to solve the MBBC problem. It inherits Bracha's diffusion mechanism: a payload message m is exchanged inside three protocol messages, SEND, ECHO, and READY. The former is initially sent by the source process to all peers, and the latter are subsequently diffused by all correct processes to all peers if certain conditions are met, namely certain quorums are reached.
The pseudo-code of $\mathcal{P}_{MBBC-RB}$ is shown in Algorithm 1. This solution overcomes the impossibility stated in Theorem 12 by leveraging on $\mathcal{O}_{FFA}$ and by fixing the round index (i.e., the moment in time) where MBBC-deliveries must occur. Every protocol's message contains the information about a specific MBBC-broadcast instance, specifically the source process label s, the message (payload) m, and the round counter r_b when the broadcast instance started. An MBBC-broadcast instance proceeds in four consecutive rounds in $\mathcal{P}_{MBBC-RB}$. In the first round r_b, the protocol's message SEND is computed by p_s and enqueued to P2P-send to all processes in the subsequent round. Every process that P2P-receives a SEND message in round r_{b+1} from p_s computes the ECHO protocol's message for $\langle s, r_b, m\rangle$ and enqueues it to P2P-send to all peers. In round r_{b+2}, the processes that receive sufficiently many ECHO messages (more than $(n+f)/2$) for an MBBC-broadcast instance from distinct peers generate the related READY protocol's message to P2P-send to all processes. Finally, in round r_{b+3}, the processes that receive a sufficient number of READY messages (more than $2f$) for an MBBC-broadcast instance from distinct peers MBBC-deliver the associated message m from p_s. An additional protocol's message with respect to Bracha [12], i.e. ABORT, is exchanged in $\mathcal{P}_{MBBC-RB}$ to guarantee the *Agreement* property in case of a faulty source. In $\mathcal{P}_{MBBC-RB}$, if a correct process p_s executes MBBC.Broadcast(m) in round r_b, then every process that is correct in round r_{b+3} triggers MBBC.Deliver(s,m) in the *compute* phase of that round; every process that is faulty in round r_{b+3} MBBC-delivers the message m from p_s at the first round $r_k > r_{b+3}$ it is correct.

We plug the fault-tolerant round counter defined by Bonnet at al. [6] inside the $\mathcal{P}_{MBBC-RB}$ protocol, enabling all correct processes to share the same value for the round index (that is assumed as an integer value). Its purpose is to fix the single round where the delivery of a certain message can take place. The round counter features are summarised in the following remark.

▶ Remark 13 (Round counter correctness [6]). In $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{BFA/FFA}\rangle$, if $n > 3f$ then every correct process p_i in round r_j stores the same value for the round index (namely the variable rc in Algorithm 1) during compute phase.

We stress the fact that protocol's messages in $\mathcal{P}_{MBBC-RB}$ (SEND, ECHO, READY, and ABORT) must be propagated in specific rounds with respect to the beginning of the MBBC-broadcast, in order to progress till the delivery of the associated message m.

A detailed description of $\mathcal{P}_{MBBC-RB}$ appears in the companion technical report [7], and we illustrate some execution examples in Appendix B, and within the proof of Lemma 14.

Algorithm 1 $\mathcal{P}_{MBBC-RB}$.

```
1: procedure INIT
2:     To_send ← ∅, Sends ← ∅, cured ← False, rc ← 1
3:     Echos ← {}, Readys ← {}, Aborts ← {}        ▷ map, ⟨s, r, m⟩ : set of process ids
4:     RC ← {}                                      ▷ map, process id : round value
5: procedure BROADCAST(m)
6:     To_send ← To_send ∪ {⟨SEND, s, rc, m⟩}
7: upon O_FFA.CURED do
8:     cured ← True
   Send Phase
9: if cured then
10:    To_send ← ∅
11: for pk ∈ To_send do
12:    for q ∈ Π do
13:        P2P.send(q, pk)
   Receive Phase
14: Sends ← ∅, Echos ← {}, Readys ← {}, Aborts ← {}, RC ← {}
15: upon P2P.deliver(q, ⟨Type, s, r_b, m⟩) do
16:    if s = q and Type = SEND then
17:        Sends ← Sends ∪ {⟨s, r_b, m⟩}
18:    if Type = ECHO then
19:        Echos[⟨s, r_b, m⟩] ← Echos[⟨s, r_b, m⟩] ∪ {q}
20:    if Type = READY then
21:        Readys[⟨s, r_b, m⟩] ← Readys[⟨s, r_b, m⟩] ∪ {q}
22:    if Type = ABORT then
23:        Aborts[⟨s, r_b, m⟩] ← Aborts[⟨s, r_b, m⟩] ∪ {q}
24: upon P2P.deliver(q, ⟨ROUND, j⟩) do
25:    RC[q] ← j
   Compute Phase
26: To_send ← ∅, rc ← GETMAJORITY(RC.VALUES)
27: for ⟨s, r_b, m⟩ ∈ Sends do
28:    if rc = r_{b+1} then
29:        To_send ← To_send ∪ {⟨ECHO, s, r_b, m⟩}
30: for ⟨s, r_b, m⟩ ∈ Echos do
31:    if |Echos[⟨s, r_b, m⟩]| > (n + f)/2 then
32:        To_send ← To_send ∪ {⟨READY, s, r_b, m⟩}
33:    else if |Echos[⟨s, r_b, m⟩]| > f then
34:        To_send ← To_send ∪ {⟨ABORT, s, r_b, m⟩}
35: for ⟨s, r_b, m⟩ ∈ Aborts do
36:    if |Aborts[⟨s, r_b, m⟩]| > f then
37:        |Readys[⟨s, r_b, m⟩] ← ∅
38: for ⟨s, r_b, m⟩ ∈ Readys do
39:    if |Readys[⟨s, r_b, m⟩]| > 2f then
40:        if ((rc = r_{b+3}) or (cured and rc > r_{b+3} and O_FFA.FAULTY_AT ≤ r_{b+3}))
           and (∄⟨s, r_k, m⟩ ∈ Readys : (|Readys⟨s, r_k, m⟩| > 2f) ∧ (r_k < r_b)) then
41:            DELIVER(s,m)
42:        To_send ← To_send ∪ {⟨READY, s, r_b, m⟩}
43: cured ← False, rc ← rc+1, To_send ← To_send ∪ {⟨ROUND, rc⟩}
```

6.2 Correctness Proofs

We remark that in S-MOB$^+$ mobile agents can move only between the *compute* and *send* phase of two consecutive rounds. This implies that Δ_s is assumed greater than or equal to one round. Such mobility model has the following effects to the agents' capabilities: at the beginning of a round r_j, mobile agents can potentially control the messages that are diffused by $2f$ processes, the ones where the mobile agents are placed in r_j and the others where they were in the previous round r_{j-1} (they can set in round r_{j-1} the messages that will be exchange by freed processes in round r_j). This capability can partially be mitigated by the local failure detector $\mathcal{O}_{FFA}$: a process can discard all messages queued to be send right after the failure detector notifies the CURED() event. It follows that, at the beginning of a round, at most f processes may not participate in the protocol and at most f may have a Byzantine behavior.

The following Lemmas and Theorem state the correctness of $\mathcal{P}_{MBBC-RB}$ in solving the MBBC problem and its fault-tolerance optimality with respect to the number of tolerated mobile agents.

▶ **Lemma 14.** *If $\Delta_b \geq 2$ rounds and $\Delta_c \geq 1$ round, then $\mathcal{P}_{MBBC-RB}$ solves the Mobile Byzantine Broadcast Channel problem (MBBC) in $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{FFA}\rangle$ if $n > 5f$.*

Proof. For simplicity, we give the proof assuming the minimum values for Δ_b and Δ_c. The arguments extend to higher values.

($\Delta_b = 2$ rounds, $\Delta_c = 1$ round)-Validity. We prove that if we assume $\Delta_b = 2$ rounds, $\Delta_c = 1$ round, and a process p_s is correct in round r_b when it executes MBBC.BROADCAST(m), then every process that is Δ_c-infinitely often correct eventually triggers MBBC.DELIVER(s,m), that implies the (Δ_b, Δ_c)-*Validity* property. The MBBC-delivery of a message m from a process p_s may occur either because p_s was correct in round r_b and executed MBBC.BROADCAST(m) or since p_s was faulty at some round $r_d < r_b$ and P2P-sent a SEND message with payload m. Let us assume that process p_s has not P2P-sent yet the SEND message with payload m neither as correct or faulty before round r_b, that it is correct in rounds r_b and r_{b+1} ($\Delta_b = 2$) and executes the procedure Broadcast with parameter m in round r_b. The $\langle$SEND, s, r_b, m$\rangle$ message is then prepared (line 6) to be relayed to all other processes (lines 11-13). In round r_{b+1}, the $\langle$SEND, s, r_b, m$\rangle$ message is P2P-sent by p_s to all processes and it is received by all but f (the ones controlled by mobile agents); it follows that $n - f$ processes executes lines 15-17 during the *receive* phase in round r_{b+1} and lines 27-29 in the *compute* phase, preparing the $\langle$ECHO, s, r_b, m$\rangle$ message to P2P-send in round r_{b+2}. In round r_{b+2}, at least $n - 2f$ processes relay the message $\langle$ECHO, s, r_b, m$\rangle$ (f process may be faulty in round r_{b+2} and f process may have been faulty in round r_{b+1}) and it is received by $n - f$ processes (again, the ones not controlled by mobile agents). These processes execute lines 15, 18 and 19 in the *receive* phase and lines 31 and 32 in the *compute* phase. In particular, the condition inside the *if* statement at line 31 is verified due to the assumption $n > 5f$, given that $n - 2f > (n + f)/2$, and line 32 is executed preparing $\langle$READY, s, r_b, m$\rangle$ message to P2P-send in round r_{b+3}. Finally, in round r_{b+3}, the same reasoning given for round r_{b+2} applies and $n - f$ processes execute lines 39-41, given $n - 2f > 2f$ and Remark 13, and thus they trigger Deliver with parameters s and m. At every round $r_j > r_{b+3}$ the $\langle$READY, s, r_b, m$\rangle$ message is P2P-sent by all the correct processes not faulty in round r_{j-1} (that are at least $n - f$). The *if* statement at line 40 guarantees that every process that was faulty in round r_{b+3} delivers message m from p_s at the first round $r_k > r_{b+3}$ it is correct. Finally, in case *(i)* process p_s was faulty and P2P-sent the SEND message with payload m in round $r_k < r_b$, *(ii)* every Δ_c-infinitely correct process MBBC-delivered m from p_s, and *(iii)*

process p_s is correct in round $r_b > r_k$ and executes MBBC.Broadcast(m), then the claim still follows: the message m has been already MBBC-delivered (further details can be found in the *Agreement* property's proof).

No duplication. The second sub-condition of the *if* statement at line 40 guarantees that the entire *if* statement is verified only for the minimum r_j among all the tuples $\langle s, *, m\rangle$ (i.e. the MBBC-delivery is independent from the r_b parameter). The first sub-condition inside the *if* statement at line 40 is verified only once among all the rounds a mobile agent does not control the process. More in detail, if the *cured* variable is False, the condition is verified only in round r_{b+3} for the tuple $\langle s, r_b, m\rangle$. Otherwise, the *if* statement in line 40 is verified in round $r_k > r_{b+3}$ when a mobile agent, arrived on the process in round $r_j \leq r_{b+3}$, leaves the process, that occurs only once on a process during the entire lifetime of the system given Remark 13. The condition $rc > r_{b+3}$ in line 40 is not required but simplifies this proof.

($\Delta_b = 2$)-Integrity. For the sake of contradiction, let us assume that a process p_i is correct in round r_k and executes MBBC.Deliver(s,m), that process p_s is correct in rounds r_b and r_{b+1} (that is, $\Delta_b = 2$), and that it does not execute MBBC.Broadcast(m) in round r_b. Process p_i MBBC-delivers m from p_s either in round $r_k = r_{b+3}$ if p_i is correct, or at the first round $r_k > r_{b+3}$ when p_i is correct. In the former case, more than $2f$ processes sent message $\langle \mathsf{READY}, \mathrm{s}, r_b, \mathrm{m}\rangle$ in round r_{b+3}, therefore more than $(n+f)/2$ processes sent message $\langle \mathsf{ECHO}, \mathrm{s}, r_b, \mathrm{m}\rangle$ in round r_{b+2}, that implies that at least $(n+f)/2-f$ processes were correct in round r_{b+1} and received $\langle \mathsf{SEND}, \mathrm{s}, r_b, \mathrm{m}\rangle$ in round r_{b+1} from p_s (lines 28-29). No procedure in $\mathcal{P}_{MBBC-RB}$ allows a correct process p_s to P2P-send $\langle \mathsf{SEND}, \mathrm{s}, r_b, \mathrm{m}\rangle$ messages except Broadcast(m). It follows that the latter scenario occurred and process p_i was faulty in round r_{b+3}. As a matter of fact, correct process p_i P2P-received more than $2f$ $\langle \mathsf{READY}, \mathrm{s}, r_b, \mathrm{m}\rangle$ messages from distinct processes in round r_k. For the same reasoning as in the former case, this implies that a correct process p_s sent $\langle \mathsf{SEND}, \mathrm{s}, r_b, \mathrm{m}\rangle$ messages but no procedure except Broadcast(m) allows it. This leads to a contradiction and the claim follows.

($\Delta_c = 1$)-Agreement. We proved, in the *Validity* proof, that this property is satisfied in the case of a correct source. Faulty processes cannot collude to make one of the *if* statements at lines 31, 33, 36 and 39 verified for a message m never sent over the P2P links of a process p_s. More in detail, the attacker cannot attempt to make any correct process MBBC-deliver a message m from p_s without compromising p_s. We prove that if p_s is faulty and P2P-sends $\langle \mathsf{SEND}, \mathrm{s}, r_b, \mathrm{m}\rangle$ messages in round r_b, then either all Δ_c-infinitely correct processes delivers m from p_s or no Δ_c-infinitely correct processes delivers m from p_s. For the sake of contradiction, let us assume that all Δ_c-infinitely often correct processes but some, $p_1, p_2, \ldots, p_i$, MBBC-delivered a message m from p_s. It follows that there is no round r_j where more than $2f$ correct processes concurrently P2P-send $\langle \mathsf{READY}, \mathrm{s}, r_b, \mathrm{m}\rangle$. This implies that the correct processes that delivered m are at most $2f$. According to the protocol, such processes receive a quorum of ECHO messages and at most f ABORT messages about m, to generate the required READY messages. More in detail, they received ECHO messages from at least $2f+1$ correct processes. At that point, the faulty processes decided which correct processes reached the quorum of ECHO messages. Nevertheless, each correct process that did not reach the quorum generated an ABORT message. It follows that at most f correct processes did not reach the quorum, whereas $n-f-f$ processes were correct and generated the READY message, which was disseminated by at least $n-3f$ of them in the subsequent round. Given that $n > 5f$, at least $2f+1$ correct processes concurrently disseminate a READY message and thus all correct processes in round r_{b+3} must MBBC-deliver it. This lead to a contradiction and the claim follows. ◀

▶ **Lemma 15.** *The Mobile Byzantine Broadcast Channel problem (MBBC) is solvable in* $\langle$*SYNC, S-MOB*$^+$*,* $\mathcal{O}_{FFA}\rangle$ *only if* $n > 5f$*.*

Proof. The claim follows by extending the results proven by Backes and Cachin [3] and by Raynal [25]. The former states that the BRB problem can be solved in a static distributed system where at most t processes may fail-stop, and at most f processes are Byzantine, if and only if $n > 3f + 2t$. Similarly, Raynal proved that the BRB problem can be solved in a static distributed system, where t_l processes may not send messages, and t_s processes may send spurious messages (processes may exhibit both behaviors during the lifetime of the system), if and only if $n > 2t_l + t_s$.

Both scenarios can be simulated by an attacker in our system: the mobile agents can continuously alternate between two disjoint sets P_1 and P_2 of f processes, namely it can turn faulty all processes in P_1 in all rounds $r_j, j \in \mathbb{N}$, and all processes in P_2 in all rounds r_{j+1}, sending spurious messages from process in P_1 and no message from peers in P_2. Therefore, all processes in P_1 send spurious messages (behaving like f Byzantine faulty processes), and all the processes in P_2 send no message (like f fail-stop faulty processes), and the claim follows. ◀

▶ **Theorem 16.** *The Mobile Byzantine Broadcast Channel problem (MBBC) is solvable in* $\langle$*SYNC, S-MOB*$^+$*,* $\mathcal{O}_{FFA}\rangle$ *with* $\mathcal{O}_{FFA}$ *if and only if* $n > 5f$*.*

Proof. It follows from Lemmas 14 and 15. ◀

The following Corollary extends the optimality of $\mathcal{P}_{MBBC-RB}$ to the case of slower agents. In other words, even if the mobile agents are slower we are not able to tolerate more agents solving MBBC.

▶ **Corollary 17.** *The Mobile Byzantine Broadcast Channel problem (MBBC) is solvable in* $\langle$*SYNC, S-MOB*$^+$*,* $\mathcal{O}_{FFA}\rangle$ *if and only if* $n > 5f$*, for each* $\Delta_s \geq 1$ *round. Furthermore, the actual value of* Δ_s *can be unknown to the processes.*

Note that MBBC and MBBR specifications do not allow processes to be *terminate*, namely to eventually stop propagating messages through the P2P primitive. Intuitively, processes need to continuously relay the messages in order to enforce Δ_c-*Totality/Agreement* and thus allow every temporarily faulty process to eventually deliver a broadcast message. Furthermore, as argued in Section 5, processes are not able to infer if a specific process has delivered a message, and thus conclude if all processes delivered a message when correct. Additional assumptions enabling termination can be considered, such as an upper-bound on the time a process becomes correct when faulty.

7 MBBC with multiple deliveries

The impossibilities identified in Section 5 arise for the general specification we defined. In fact, alternative or weaker specifications could be implementable under weaker assumptions. More in detail, we proved that no protocol can solve the MBBC in $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{BFA}\rangle$. We therefore investigate the possibility of a weaker primitive that can be realized when the stringent conditions identified in Theorem 16 are not satisfied.

We start by considering the case where no local failure detector is available, that is, the case of $\mathcal{O}_{NFA}$. The following Theorem show that a weaker MBBC primitive, where the *No duplication* property is not satisfied, is realizable in $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{NFA}\rangle$.

▶ **Theorem 18.** *A weaker Mobile Byzantine Broadcast Channel primitive, not guaranteeing the* No duplication *property, is realizable in* $\langle$`SYNC, S-MOB`$^+$`,` $\mathcal{O}_{NFA}\rangle$ *if* $\Delta_b = 2$ *rounds,* $\Delta_c = 1$ *round, and* $n > 6f$*.*

Proof. Let us consider the $\mathcal{P}_{MBBC-RB}$ protocol defined in Algorithm 1. Let us ignore the lines that interacts with the local failure detector, namely 7, 8 and 40. Let us substitute all the occurrences of parameter f with $\bar{f} = 2f$ in Algorithm 1.

The difference with respect the setting considered in Lemma 14 is that processes are not aware of being compromised. In particular, they may diffuse messages with P2P-links previously generated by mobile agents. As a matter of fact, the protocol is restored right after the mobile agent left the process.

The proof follows from the same reasoning stated in Lemma 14 except for *No duplication* considering $\bar{f}$ instead of f in Algorithm 1. ◀

The following theorem show that having a slightly better oracle about failures, namely $\mathcal{O}_{BFA}$, permits to withstand more Byzantine agents, for the same weaker problem that does *not* guarantees no duplication.

▶ **Theorem 19.** *A weaker Mobile Byzantine Broadcast Channel primitive, not guaranteeing the* No duplication *property, is realizable in* $\langle$`SYNC, S-MOB`$^+$`,` $\mathcal{O}_{BFA}\rangle$ *if* $\Delta_b = 2$ *rounds,* $\Delta_c = 1$ *round, and* $n > 5f$*.*

Abandoning the *No duplication* guarantee, the number of message delivered becomes unbounded: the following theorem shows that it is not possible to bound the number of duplicate messages that are delivered, even assuming an intermediate oracle, namely $\mathcal{O}_{BFA}$.

▶ **Theorem 20.** *Given a constant* $\bar{k} \in \mathbb{N}^+$*, it is not possible to define a weaker Mobile Byzantine Broadcast Channel primitive, not guaranteeing the* No duplication *property, in* $\langle$`SYNC, S-MOB`$^+$`,` $\mathcal{O}_{BFA}\rangle$ *where a message* m *MBBC-Broadcast by a process* p_s *is MBBC-Delivered by a process* p_i *at most* $\bar{k}$ *time when correct.*

▶ **Corollary 21.** *Suppose a solution to a weaker Mobile Byzantine Broadcast Channel primitive, not guaranteeing the* No duplication *property, in* $\langle$`SYNC, S-MOB`$^+$`,` $\mathcal{O}_{BFA}\rangle$*. If a process* p_i *gets faulty and correct* k *times after the MBBC-Broadcast of a message* m *from* p_s*, then* p_i *MBBC-Delivers* m *from* p_s *at least* k *times.*

▶ **Theorem 22.** *Suppose a solution to a weaker Mobile Byzantine Broadcast Channel primitive, not guaranteeing the* No duplication *property, in* $\langle$`SYNC, S-MOB`$^+$`,` $\mathcal{O}_{NFA}\rangle$*. If a process* p_s *MBBC-Broadcast a message* m*, then every process* p_i *must MBBC-Deliver* m *from* p_s *infinitely often.*

8 Conclusion

We provided a specification for the Byzantine Reliable Broadcast and Byzantine Broadcast Channel problems in distributed systems affected by mobile Byzantine faults. We identified some impossibilities; in particular, we showed that both speed constraints on the mobile agents and timing assumptions on the system evolution are required to solve the problems under investigation, and we proved that the Byzantine Reliable Broadcast cannot be solved even in one of the most constrained mobile Byzantine failure models presented so far. The Byzantine Broadcast Channel problem proved to be solvable, assuming a stronger local failure detector than the ones previously considered in the literature. Lastly, we investigated a weaker Byzantine Broadcast Channel primitive, not guaranteeing the *No duplication* property,

in settings equivalent to the ones assumed in related works. Our results characterise the solvability of a fundamental problem in a general dynamic process failure model, and open the path for research on additional important tasks. In particular, to understand the gap that exists between the theoretical model (assumed in this and in related work [4, 6, 17, 24, 31, 27]) and the practical world, investigating the feasibility of the oracles and defining solutions that are as practical as possible. Furthermore, it may be interesting to relax the assumptions of instantaneous fault detection and recovery (of the protocol), to investigate whether the assumption of digitally signed messages has an impact on the solvability of the considered problems, and to analyse the Mobile Byzantine Channel problem assuming the `S-MOB` agent mobility model (which we have left open for analysis and we conjecture its solvability).

References

1 Ittai Abraham, Ling Ren, and Zhuolun Xiang. Good-case and bad-case latency of unauthenticated byzantine broadcast: A complete categorization. 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 5:1–5:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.OPODIS.2021.5.

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 Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25–29, 2022*, pages 399–417. ACM, 2022. doi:10.1145/3519270.3538475.

3 Michael Backes and Christian Cachin. Reliable broadcast in a computational hybrid model with byzantine faults, crashes, and recoveries. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 37–46. IEEE Computer Society, 2003. doi:10.1109/DSN.2003.1209914.

4 Nazreen Banu, Samia Souissi, Taisuke Izumi, and Koichi Wada. An improved byzantine agreement algorithm for synchronous systems with mobile faults. *International Journal of Computer Applications*, 43(22):1–7, 2012.

5 Romain Boichat and Rachid Guerraoui. Reliable and total order broadcast in the crash-recovery model. *J. Parallel Distributed Comput.*, 65(4):397–413, 2005. doi:10.1016/J.JPDC.2004.10.008.

6 François Bonnet, Xavier Défago, Thanh Dang Nguyen, and Maria Potop-Butucaru. Tight bound on mobile byzantine agreement. *Theor. Comput. Sci.*, 609:361–373, 2016. doi:10.1016/J.TCS.2015.10.019.

7 Silvia Bonomi, Giovanni Farina, and Sébastien Tixeuil. Reliable Broadcast despite Mobile Byzantine Faults, nov 2023. URL: https://hal.science/hal-04277831.

8 Silvia Bonomi, Antonella Del Pozzo, and Maria Potop-Butucaru. Tight self-stabilizing mobile byzantine-tolerant atomic register. In *Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, January 4-7, 2016*, pages 6:1–6:10. ACM, 2016. doi:10.1145/2833312.2833320.

9 Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Approximate agreement under mobile byzantine faults. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*, pages 727–728. IEEE Computer Society, 2016. doi:10.1109/ICDCS.2016.68.

10 Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal mobile byzantine fault tolerant distributed storage: Extended abstract. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 269–278. ACM, 2016. doi:10.1145/2933057.2933100.

11 Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal storage under unsynchronized mobile byzantine faults. In *36th IEEE Symposium on Reliable Distributed Systems, SRDS 2017, Hong Kong, Hong Kong, September 26-29, 2017*, pages 154–163. IEEE Computer Society, 2017. doi:10.1109/SRDS.2017.20.

12 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

13 Harry Buhrman, Juan A. Garay, and Jaap-Henk Hoepman. Optimal resiliency against mobile faults. In *Digest of Papers: FTCS-25, The Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, California, USA, June 27-30, 1995*, pages 83–88. IEEE Computer Society, 1995. doi:10.1109/FTCS.1995.466995.

14 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.

15 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

16 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. URL: http://www.cs.bgu.ac.il/%7Edolev/book/book.html.

17 Juan A. Garay. Reaching (and maintaining) agreement in the presence of mobile faults (extended abstract). In Gerard Tel and Paul M. B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 – October 1, 1994, Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 1994. doi:10.1007/BFB0020438.

18 Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic byzantine reliable broadcast. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 23:1–23:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.OPODIS.2020.23.

19 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. Scalable byzantine reliable broadcast. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 22:1–22:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.DISC.2019.22.

20 Damien Imbs and Michel Raynal. Trading off t-resilience for efficiency in asynchronous byzantine reliable broadcast. *Parallel Process. Lett.*, 26(4):1650017:1–1650017:8, 2016. doi:10.1142/S0129626416500171.

21 Vasilis P. Koutras and Agapios N. Platis. Chapter 3: Software rejuvenation: Key concepts and granularity. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*, pages 321–322. IEEE, 2020. doi:10.1109/ISSREW51248.2020.00092.

22 Jing Li, Tianming Yu, Ye Wang, and Roger Wattenhofer. Dynamic byzantine broadcast in asynchronous message-passing systems. *IEEE Access*, 10:91372–91384, 2022. doi:10.1109/ACCESS.2022.3202627.

23 Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *J. Netw. Comput. Appl.*, 36(1):16–24, 2013. doi:10.1016/J.JNCA.2012.09.004.

24 Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In Luigi Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 51–59. ACM, 1991. doi:10.1145/112600.112605.

25 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems – An Algorithmic Approach*. Springer, 2018. doi:10.1007/978-3-319-94141-7.

26 Michel Raynal. On the versatility of bracha's byzantine reliable broadcast algorithm. *Parallel Process. Lett.*, 31(3):2150006:1–2150006:9, 2021. doi:10.1142/S0129626421500067.

27 Rüdiger Reischuk. A new solution for the byzantine generals problem. *Inf. Control.*, 64(1-3):23–42, 1985. doi:10.1016/S0019-9958(85)80042-5.

28 Luís E. T. Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems, Taipei, Taiwan, April 10-13, 2000*, pages 288–295. IEEE Computer Society, 2000. doi:10.1109/ICDCS.2000.840941.

29 Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 57–64. IEEE, 2015. doi:10.1109/TRUSTCOM.2015.357.

30 Dimitris Sakavalas and Lewis Tseng. Delivery delay and mobile faults. 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.2018.8548345.

31 Toru Sasaki, Yukiko Yamauchi, Shuji Kijima, and Masafumi Yamashita. Mobile byzantine agreement on arbitrary network. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems – 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings*, volume 8304 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2013. doi:10.1007/978-3-319-03850-6_17.

32 Lewis Tseng. An improved approximate consensus algorithm in the presence of mobile faults. In Paul G. Spirakis and Philippas Tsigas, editors, *Stabilization, Safety, and Security of Distributed Systems – 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, volume 10616 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2017. doi:10.1007/978-3-319-69084-1_8.

A The Byzantine Reliable Broadcast and Channel Problems Specification [12, 14]

The Byzantine Reliable Broadcast and the Byzantine Broadcast Channel problems aim at specifying a communication primitive, respectively BRB and BBC, exposing two operations, BRB/BBC-BROADCAST(m) and BRB/BBC-DELIVER(s, m), where m is a message and s is a process identifier.

The BRB primitive enables all correct processes of a distributed system to agree on a single message diffused by a (potentially faulty) particular process, the source. The BBC primitive extends BRB allowing all processes to diffuse an arbitrary number of messages so that all correct processes eventually deliver the same set of messages. We say that a process p_i "BRB/BBC-broadcasts a message m" when it invokes BRB/BBC-BROADCAST(m), and p_i "BRB/BBC-delivers a message m from p_s" when it manage the BRB/BBC-DELIVER(s, m) event.

We remark that both BRB and BBC primitives assume a *static process failure model* where every process is permanently correct or faulty.

A.1 Byzantine Reliable Broadcast (BRB)

The BRB communication primitive guarantees the following properties:

- *Validity*: If a correct process p_s BRB-broadcasts a message m, then every correct process eventually BRB-delivers m from p_s.
- *No duplication*: Every correct process BRB-delivers at most one message from p_s.
- *Integrity*: If some correct process BRB-delivers a message m from p_s and process p_s is correct, then m was previously BRB-broadcast by p_s.

- *Consistency*: If some correct process BRB-delivers a message m from p_s and another correct process BRB-delivers a message m' from p_s, then $m = m'$.
- *Totality*: If some message is BRB-delivered by any correct process, every correct process eventually BRB-delivers a message.

A.2 Byzantine Broadcast Channel (BBC)

The BBC communication primitive guarantees the following properties:

- *Validity*: If a correct process p_s BBC-broadcasts a message m, then every correct process eventually BBB-delivers m from p_s.
- *No duplication*: No correct process BBC-delivers a message m from p_s more than once.
- *Integrity*: If some correct process BBC-delivers a message m from p_s and process p_s is correct, then m was previously BBC-broadcast by p_s.
- *Agreement*: If some correct process BBC-delivers a message m from p_s then every correct process eventually delivers message m from p_s.

B $\mathcal{P}_{MBBC-RB}$ execution examples

We detail in this Section several execution examples for the $\mathcal{P}_{MBBC-RB}$ protocol defined in Section 6. Given what claimed in Theorem 16, we assume that the correctness conditions for our protocol, i.e. a $\langle$SYNC, S-MOB$^+$, $\mathcal{O}_{FFA}\rangle$ system and $n > 5f$, are satisfied in all of the provided examples. We detail one example where the source is correct and two in which the source is faulty.

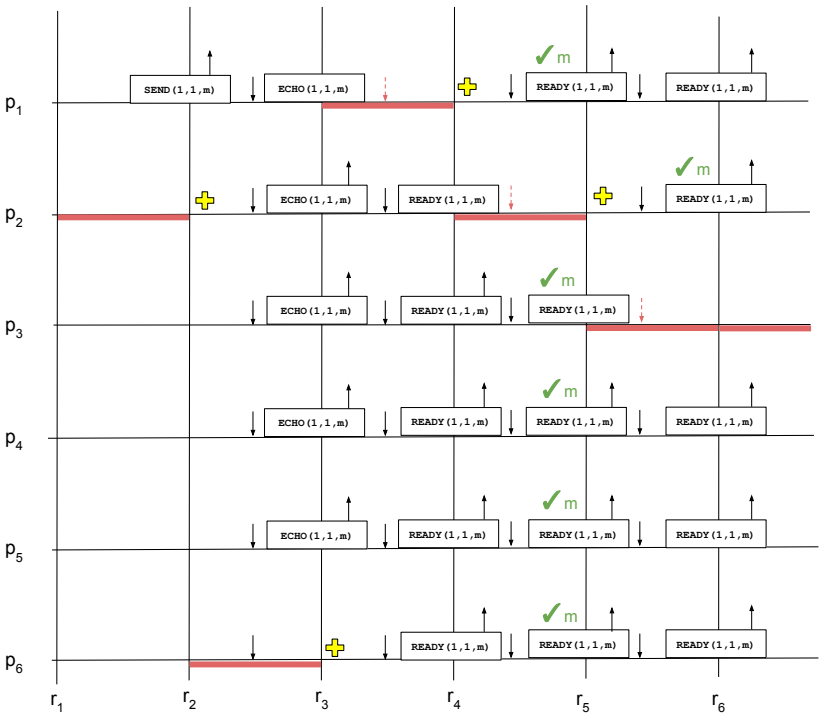


Figure 2 An execution of $\mathcal{P}_{MBBC-RB}$ with a correct source and $f = 1$.

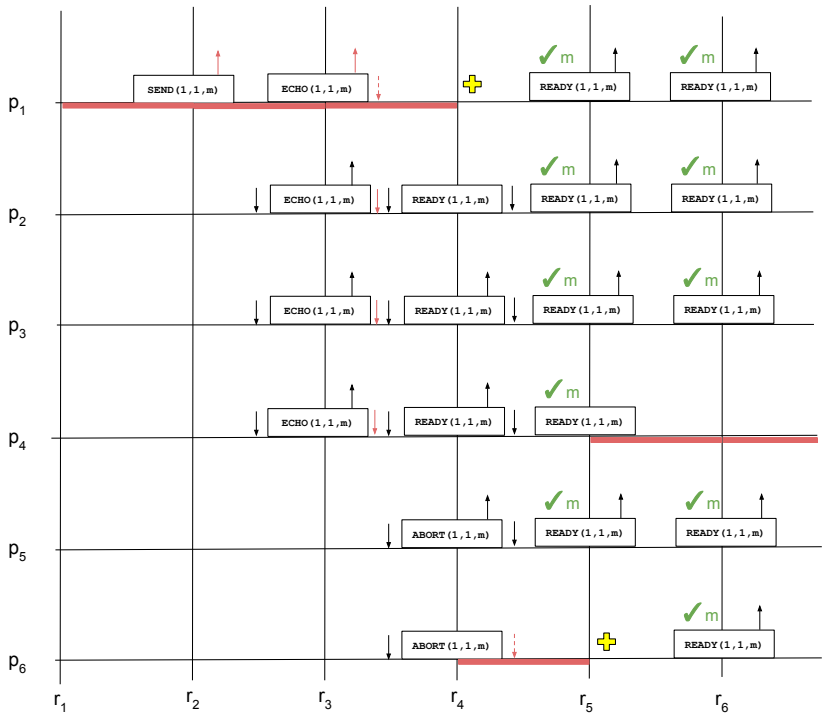


Figure 3 An execution of $\mathcal{P}_{MBBC-RB}$ with a faulty source, $f = 1$ and all infinitely often correct processes delivering.

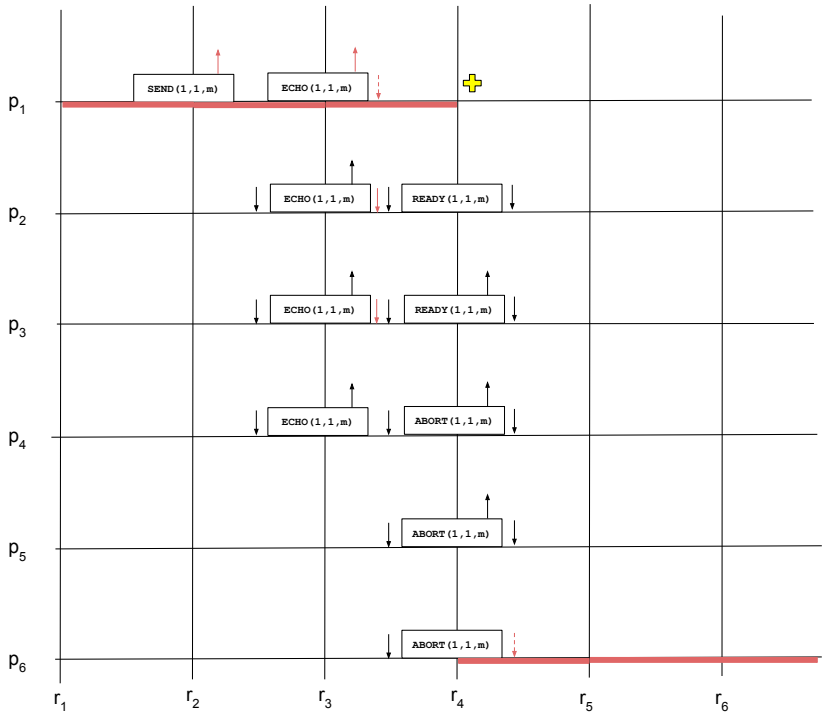


Figure 4 An execution of $\mathcal{P}_{MBBC-RB}$ with a faulty source, $f = 1$ and no infinitely often correct process delivering.

In the execution example in Figure 2, the correct source p_1 starts the MBBC-Broadcast preparing the related SEND message in round r_1, that is P2P-sent to all processes in round r_2 ($\Delta_b = 2$). Process p_2 is faulty in round r_1, then the mobile agent moves to process p_6 in round r_2. All processes but f are correct in round r_2, thus they receive the SEND message from p_1 and generate the related ECHO message. Such message is then P2P-sent to all peers by at least $n - 2f$ processes during the *send* phase in round r_3 (at most f processes could have been faulty in round r_2, p_6 in our example, and at most f processes could become faulty in round r_3, p_1 in our example where the mobile agent moves in round r_3). It follows that $n - f$ processes reach the quorum of ECHO messages generating the related READY message. Again, at least $n - 2f$ processes are correct in round r_4, P2P-send the READY message and deliver the associated payload from p_1, m, during the *compute* phase of the same round. The processes that were faulty in round r_4, p_2 in our example, deliver the message at the first round $r_k > r_4$ they get correct, because all processes that are correct in a round $r_j > r_4$ diffuse the associated READY message.

The only MBBC property that mobile agents may attempt to invalidate in a execution of $\mathcal{P}_{MBBC-RB}$ is the *Agreement* property: the *No duplication* is guaranteed by the *if* statement at line 40 in Algorithm 1 and both *Validity* and *Integrity* consider a correct source. Any source must P2P-send a well-formed SEND message (i.e., with valid source id and round label) to make a correct process proceed in the protocol to deliver a payload m. If the SEND message is P2P-sent to all correct processes, then all Δ_c-infinitely often correct processes will eventually deliver m, as shown in the previous execution, satisfying the MBBC specification. It follows that a Byzantine source must not P2P-send the SEND message to some processes. This behavior has two possible outcomes in our protocol: either all correct processes MBBC-deliver the diffused message or no correct process does it. Let us assume that the mobile agent commands p_1 to P2P-send the SEND message to $\lfloor (n - f)/2 \rfloor - f$ processes, in order to control which ones will proceed in the $\mathcal{P}_{MBBC-RB}$ protocol generating the READY message in round r_4.

In the execution depicted in Figure 3, process p_1 is a faulty source that attempts to prevent the *Agreement* property of MBBC from being satisfied. Specifically, it P2P-sends the ECHO message only to part of the processes, process p_2, p_3, and p_4, that reach the quorum required to generate the READY message. In this case, processes p_5 and p_6 generate the ABORT message but only f of them, namely p_5, P2P-send it, thus blocking no correct process from proceeding in the MBBC-delivery of m from p_1. Nonetheless, in this case more than $2f$ processes are correct and P2P-send the READY message in round r_4. It follows that all Δ_c-infinitely often correct processes eventually deliver the associated payload m.

Differently from the previous example, in the execution in Figure 4 process p_1 sends the ECHO message to processes p_2 and p_3. It follows that all other correct processes, p_4, p_5, and p_6, generate the ABORT message. At most f of them, process p_6 in the example, can be blocked from P2P-sending the ABORT message. It follows that more than f processes diffuse to all correct ones the ABORT message and thus no process delivers the associated payload m. It follows that the specification is not violated in such execution.

Probable Approximate Coordination

Ariel Livshits
Yahoo! Research, Haifa, Israel

Yoram Moses
Technion, Haifa, Israel

Abstract

We study the problem of how to coordinate the actions of independent agents in a distributed system where message arrival times are unbounded, but are determined by an exponential probability distribution. Asynchronous protocols executed in such a model are guaranteed to succeed with probability 1. We demonstrate a case in which the best asynchronous protocol can be improved on significantly. Specifically, we focus on the task of performing actions by different agents in a linear temporal order – a problem known in the literature as *Ordered Response*. In asynchronous systems, ensuring such an ordering requires the construction of a message chain that passes through each acting agent, in order. Solving *Ordered Response* in this way in our model will terminate in time that grows linearly in the number of participating agents n, in expectation. We show that relaxing the specification slightly allows for a significant saving in time. Namely, if *Ordered Response* should be guaranteed with high probability (arbitrarily close to 1), it is possible to significantly shorten the expected execution time of the protocol. We present two protocols that adhere to the relaxed specification. One of our protocols executes exponentially faster than a message chain, when the number of participating agents n is large, while the other is roughly quadratically faster. For small values of n, it is also possible to achieve similar results by using a hybrid protocol.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed coordination, ordered response, exponentially distributed delay

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.19

Related Version *Extended Version*: `https://arxiv.org/pdf/2311.05368.pdf` [16]

Funding This work was supported in part by the Israel Science Foundation under grant 2061/19.

Acknowledgements Yoram Moses is the Israel Pollak academic chair at the Technion. The authors would like to thank Yonathan Shadmi for his most valuable help.

1 Introduction

Numerous applications of distributed systems rely on promptly responding to spontaneously occurring events triggered by the environment. These events can range from the activation of fire alarms or smoke detectors to transactions involving bank account deposits or withdrawals. In order to ensure that the system behaves correctly, it may be necessary to execute one or more relevant actions. The sequential execution of these actions often holds significance, particularly when dealing with financial transactions. Typically, a diverse range of such actions, including condition testing and related updates, must be carried out to successfully complete these transactions.

This paper considers solutions to a distributed coordination problem called *Ordered Response* (*OR*). In *Ordered Response*, agents must perform individual actions in a particular linear order, in response to the spontaneous arrival of an external input to the system. For example, let there be a system with three agents i_1, i_2 and i_3. Let t_1, t_2 and t_3 be the points in time at which the agents perform their respective actions. Then it must hold that $t_1 \leq t_2 \leq t_3 < \infty$ in every run of a protocol that solves *OR*. Notice that if the agents act simultaneously, then the specifications of *OR* are satisfied.

The *Ordered Response* problem has been analyzed previously in the contexts of synchronous and asynchronous message-passing communication models [3, 5]. However, in many modern day multi-agent systems, message delays are subject to probabilistic bounds, i.e., determined according to a probabilistic distribution. Examples can be seen in wireless sensor networks (WSNs). A WSN is a distributed multi-agent system that is often composed of a large quantity of disposable sensors that communicate using wireless broadcast. Messages sent in this network experience random delays due to random changes in the environment. However, it was observed in [1] that a single server M/M/1 queue [12] can represent the cumulative link delay in a WSN, when the random delays are modeled as exponential random variables.

In this work, we investigate the *Ordered Response* problem in a model where message arrival times are unbounded, but are sampled from an exponential probability distribution.[1] We are particularly interested in protocols that achieve correctness with high probability (w.h.p.) and terminate in a short expected time. It is unclear, based on initial observations, whether the previous solutions are the most optimal for this particular task.

One of the conclusions that can be drawn from previous work done by Chandy & Misra [5] is that ensuring a linear ordering of actions at distinct sites of an asynchronous system necessitates the existence of a message chain among the coordinating agents. Namely, in order for agents to ensure that their respective actions are performed in a linear temporal order, a message chain visiting each of the sites in this order must be formed. Concretely, every asynchronous protocol that solves *Ordered Response* must construct a message chain among the agents. Consequently, in an asynchronous system of n agents, *Ordered Response* can only be solved in time that is linear in n, since a chain of at least $n-1$ messages must be sent.

Since message delays are unbounded, our probabilistic model appears to be closer to an asynchronous model, than a synchronous model. However, the assumption of exponentially distributed delays may be leveraged to significantly shorten the expected time of termination. Intuitively, since they naturally induce a distribution on the runs of a given protocol, it is possible to choose appropriate waiting times for the agents such that correct runs are more likely to happen. For example, say an agent broadcasts a message to 99 other agents. Assuming that arrival times are independent and identically distributed exponential random variables with parameter 1 $[sec^{-1}]$, then, with probability 0.999, all messages will arrive within roughly 11.52 seconds. So, assuming that every agent has access to a global clock and that the broadcast happened at time $t = 0$, after 12 seconds all agents may act simultaneously, achieving Ordered Response with high probability. In contrast to a message chain, where the expected response time would be at best 99 seconds, a significant improvement.[2]

The main contributions of this paper are:

- A study of *Ordered Response* protocols in the Exponentially Distributed Delay (EDD) model, which we are the first to formally define.
- Two tunable *Ordered Response* protocols are presented. Both solve the problem with probability as close to 1 as desirable. The first assumes the existence of a global clock and executes **exponentially** faster than a message chain, when the number of participating agents n is large. The second does not use a global clock, and instead employs a

[1] For modelling WSN link delays, the most widely used distributions are Gaussian, exponential, gamma and Weibull [15, 20]. We opted for the exponential distribution for its simplicity in analyzing the model, as considering other distributions would introduce technical challenges beyond the scope of our work.

[2] We assume that agents can send only a constant amount of messages for reasons we will go into later on.

procedure that ensures that, with high probability, the local clocks of the agents are *probably approximately* synchronized. As a result, it executes slower than the first former protocol, but is still quadratically faster than the message chain protocol.

- For each of the above protocols, we also provide a hybrid version that combines the protocol with a message chain to efficiently achieve *Ordered Response* w.h.p. for any n.

2 Related Work

The *Ordered Response* problem was introduced by Ben-Zvi & Moses in [3], and was investigated there in the context of the Bounded Communication Model, in which message delays have (deterministic) upper bounds. Given such bounds, the mere passage of time can provide information about the occurrence of events at remote sites, without the need for explicit confirmation. The authors extend Lamport's *happened-before* relation [13] to define a causal structure called the "centipede" (a strict generalization of the "message chain"), and show that centipedes must exist in every execution in which linear ordering of actions is ensured. The concurrent Ordered Response protocols presented in this work implement a similar communication approach to the centipede by broadcasting information. Our model does not assume deterministic upper bounds on message delays, but rather probabilistic ones. The result is a protocol that solves Ordered Response with high probability.

Assuming an exponential distribution on message arrival times is not new. In fact, such assumptions are frequently made in the context of Wireless Sensor Networks (WSNs). Specifically, for the problem of estimating and/or bounding clock skew of sensors in WSNs [1, 6, 10, 11, 14, 21]. It was observed in [1] that a single server M/M/1 queue can represent the cumulative link delay in a WSN, when the random delays are modeled as exponential random variables. Moreover, the Minimum Link Delay algorithm, proposed in [1], which shows good performance, was derived independently in [10] assuming exponentially distributed network delays. Therefore, the assumption of an exponential distribution seems to be suited for modeling WSN link delays.

In the following, we will introduce a Concurrent Ordered Response protocol designed for an environment lacking a global clock. This protocol employs a procedure to achieve probable approximate synchronization of the agents' local clocks. Previous literature has proposed clock synchronization protocols in environments with probabilistic message delays [2, 8, 18, 19]. These protocols, similar to our solutions, guarantee clock synchronization with probability as close to 1 as desirable. However, they rely on agents being able to remotely read another agent's local clock multiple times to accomplish this. The *novelty* of our approach lies in its minimal communication requirements.[3] Agents broadcast messages only once (or twice in the hybrid variant) in order to achieve probable approximate synchronization. Along with the subsequent second phase, the agents successfully achieve Ordered Response with high probability.

The rest of this work is organized as follows. Section 3 presents our model and existing *Ordered Response* solutions in the synchronous and asynchronous models. Section 4 presents the CORE protocol when a global clock is present in the model. As well as, a hybrid protocol of CORE with a message chain. Section 5 details a procedure to *probably approximately* synchronize local clocks of agents, and how we use it in the CORE protocol when a global clock is not present in the model. Finally, Section 6 concludes this work with our conclusions.

[3] Minimal communication is crucial in WSNs due to battery life concerns.

3 Preliminaries

3.1 Model Formulation

In the Ordered Response problem, we consider a set of $n+1$ agents $\mathbb{P} \triangleq \{i_0, i_1, \ldots i_n\}$. The agents are assumed to be connected via a complete communication network, and agents can communicate with one another by sending and receiving messages (including broadcasts) that contain any form of information. We assume that the value of n is available as an input to the protocols, and every agent is aware of its own ID, as well as the IDs of all the other agents in the system. The system is considered to be event-driven, i.e., agents change their state only when some local event occurs. Agents wait between the occurrence of these events, and are assumed inactive prior to receiving the first message or external input from the environment in an execution.

We assume that all agents can measure the passage of time accurately using local clocks. A clock can be used to set a timer that will cause the agent to activate at some point in the future. Hence, any local event experienced by an agent is either the receipt of a message or a timer signalling. All agents are assumed to be inactive up until the first local event takes place. When an agent is activated, it immediately performs some local calculation and/or action as dictated by a deterministic distributed protocol that is shared among all agents in the system. We assume the agents are well-behaved and reliable, i.e., agents do not crash or disobey the protocol.

We designate agent i_0 to be a "supervisor" agent. At time $t = 0$, the agent i_0 receives an external input from the environment that triggers the system into action. Each of the other "worker" agents i_k has a special action α_k unique to itself. When the external input is delivered, the supervisor begins the process by sending messages to a subset of agents. From then on, each of the agents, once active, coordinates with the others with the goal of performing their respective actions in a *linear temporal order*.

Formally, let $\mathcal{P}$ be some shared protocol, and denote the set of runs of $\mathcal{P}$ by $R(\mathcal{P})$. We say that a run $r \in R(\mathcal{P})$ adheres to the specifications of *Ordered Response* if the following two properties hold:

- *Safety:* $\forall k > 1$: agent i_k does not perform the action α_k before agent i_{k-1} performs the action α_{k-1},
- *Liveness:* every agent i_k eventually performs action α_k.

Accordingly, we say that $\mathcal{P}$ solves *Ordered Response* if every run $r \in R(\mathcal{P})$ of the protocol adheres to the *Ordered Response* specifications. Later, we will investigate protocols that do not solve *Ordered Response*, but rather solve it with high probability. We say that a protocol $\mathcal{P}$ solves *Ordered Response* with some probability p, if the probability that a run of the protocol will satisfy both *Safety* and *Liveness* is at least p.

3.2 A Probability Measure on Runs

Unlike in classical message-passing communication models, we assume that message delays are determined by a probability distribution. Specifically, the arrival times of messages that are sent over any communication channel, are assumed to be sampled from an exponential distribution with parameter λ. The delays of distinct messages (including those that are sent via broadcast) are assumed to be statistically independent of one another. We name this communication model the Exponentially Distributed Delay model, denoted by the abbreviation EDD, which is shorthand for EDD(λ) when λ is clear from the context.

In this work, we exclusively consider deterministic protocols, and so the communication infrastructure provides the only source of randomness. The exponential distribution on message delays induces a probability distribution on the set of runs of any given protocol $\mathcal{P}$. We map every run of a protocol $\mathcal{P}$ in the EDD model, to a sampling of a countably infinite set of exponential random variables with parameter λ. Every exponential random variable in this set represents the delay of a message sent in a run of the protocol. This induces a natural probability space $\mathcal{S} = (\Omega, \mathcal{F}, \Pr)$ that we use to prove our claims (see Appendix A for a complete exposition of the probability space).

In a probabilistic environment, a simplistic solution to the Ordered Response problem may involve repeatedly transmitting the same message between agents. The intention is to increase the likelihood of an earlier message arrival beyond what a single message's probability distribution allows. This approach increases the message complexity of the protocol and, perhaps more crucially, imposes considerable energy costs on the individual agents, which is often unreasonable, e.g., in WSNs. Therefore, we assume any agent can only transmit a constant number of messages (i.e., independent of the number of agents in the system). If a protocol does make use of a constant number of retransmissions of the same message to the same destination (say k such messages), we can effectively model this as a single message that is sampled from an exponential distribution with a larger parameter ($\tilde{\lambda} \geq k\lambda$).

3.3 Performance and Correctness Metrics

In this work, we investigate protocols that ensure *Ordered Response* with high probability. Specifically, we allow protocols that ensure a linear temporal order with some probability p. A run in which both the *Safety* and *Liveness* conditions hold we call a *correct run*. Formally, let $t_k : \Omega \to \mathbb{R}$ be a random variable that represents the time that agent i_k performs α_k. The value of t_k is determined by the protocol $\mathcal{P}$ and a realization $\omega \in \Omega$, i.e., for every agent i_k, the protocol $\mathcal{P}$ induces a function τ_k such that $t_k = \tau_k(\omega)$. The probability of a correct run of $\mathcal{P}$ is then given by $p \triangleq \Pr(t_1 \leq t_2 \leq \ldots \leq t_n < \infty)$.

We compare protocols using two performance measurements that are a function of the number of agents n. The first is message complexity, as in the number of messages sent, and the second is *response time*. The response time of an Ordered Response protocol in a given run is defined as the time at which the last agent performs its unique action. Thus, the response time of the protocol is a random variable $RT \triangleq \max\{t_1, \ldots, t_n\}$. Since message delays are unbounded in the EDD model, there exist runs of the protocol that never terminate. Although the probability measure of these runs is zero, measuring the response time of protocols in this model is meaningless. Instead, we measure the *expected* response time, denoted by $\mathbb{E}[RT]$.

3.4 Asynchronous Ordered Response and the Message Chain Protocol

In an asynchronous system, there is no global clock and no bound on the arrival time of messages. Fortunately, there is a very simple Ordered Response protocol in this model as well, called the Message Chain protocol. At time $t = 0$, the supervisor i_0 sends a message to i_1. For all $k \geq 1$, when i_k receives a message from i_{k-1} it performs α_k and sends a message to agent i_{k+1} (if $k < n$). This creates a message chain among agents.

Since communication in the asynchronous setting is reliable, the message chain protocol ensures *Safety* and *Liveness* in every run. The correctness of the protocol stems from the fact that no agent acts before receiving a direct message from its predecessor in the agent ordering. Notice that a direct message is not a necessary condition for an agent to act.

Rather, as the analysis in Chandy & Misra [5] suggests, a message chain between every two consecutive agents in the ordering is required to ensure a linear temporal ordering of actions. Therefore, the message chain protocol presented above is an optimal solution for the asynchronous model in terms of both message complexity and response time.

To measure the response time of a protocol in an asynchronous system, we consider each message arrival time to be of length of one time unit.[4] Under such accounting, the message chain protocol exhibits $\Theta(n)$ response time, due to its sequential behavior. Since the Message Chain protocol is the optimal Ordered Response protocol in an asynchronous system, this is also a lower bound for the asynchronous Ordered Response problem, in general.

3.5 Ordered Response with Probability 1

While message arrival times are not deterministically bounded in the EDD model, they are bounded in the probabilistic sense. In particular, messages are more likely to arrive sooner rather than later. Our focus in this work will be the design and analysis of Ordered Response protocols that achieve good performance in the EDD model.

We consider the Message Chain protocol as our baseline. Notice that the protocol ensures that the probability of a *correct run* is 1. The reason is that the protocol ensures *Liveness* only *w.p.* 1, due to the fact that messages in the EDD model arrive in finite time *w.p.* 1.

▶ **Definition 1.** *We say that a run $r \in R(\mathcal{P})$ contains a message chain, if there exist a set of n messages $m_1, m_2, \ldots, m_n$ that are sent in r, and for all $1 \leq k \leq n-1$ the agent i_k sends m_{k+1} only after it receives m_k from agent i_{k-1}.*

Additionally, in a run that contains a message chain, we say that an agent i_k "receives a message chain of length k at time t" if at time t agent i_k receives message m_k.

▶ **Theorem 2.** *In the EDD model, if every run of a protocol $\mathcal{P}$ contains a message chain, then it holds for $\mathcal{P}$ that $E[RT] \in \Omega(\frac{n}{\lambda})$.*

Proof. If the run implements a message chain, then agent i_{n-1} does not terminate prior to receiving a message chain of length $n-1$, since it has to send the final message m_n. By definition, the response time of a protocol in a given run is greater or equal to the time at which i_{n-1} terminates. Thus, $RT > \xi$ where ξ is a sum of $n-1$ *i.i.d.* exponential random variables with parameter λ. Hence, $\mathbb{E}[RT] > \mathbb{E}[\xi] = \frac{n-1}{\lambda}$. ◀

From Theorem 2 we conclude that the expected response time of the Message Chain protocol in the EDD model is in $\Omega(\frac{n}{\lambda})$. Later on, we investigate protocols that trade off probability of correctness for expected response time that is sub-linear in the number of participating agents n.

4 Ordering with a Global Clock

In order to investigate the design of faster protocols, we relax the specification slightly to allow the protocol to guarantee *Safety* and *Liveness* properties with high probability (*w.h.p.*). More precisely, we now consider protocols that solve Ordered Response with probability $0 < p < 1$, where p can be as close to 1 as desirable. Our goal is to find a protocol that significantly outperforms the Message Chain protocol, in terms of expected response time.

[4] This is generally the accepted approach in the literature for measuring time of execution of a run in an asynchronous system [17].

The main disadvantage of the Message Chain protocol is its sequential behaviour. However, in the EDD model, messages are more likely to arrive earlier rather than later. Our approach leverages this towards speeding-up the response time of the protocol. The main idea behind the protocols that will be described in this section is the concurrent transmission of information to all parties. For example, instead of sending only one message to one agent informing him that the external input has arrived, the supervisor broadcasts this information to all agents in the system. We call this approach – Concurrent Ordered REsponse, abbreviated by CORE.

4.1 The CORE Protocol

In a synchronous system, there usually exists a deterministic upper bound Δ on the arrival time of messages. In the CORE protocol there is a parameter $\mathcal{D}$ that is configurable, rather than being a constant defined by the environment. This parameter is chosen so that *w.h.p.* all the messages that are sent in the supervisor's broadcast are delivered within time $\mathcal{D}$. The psuedocode of the protocol is presented in Algorithm 1.

Algorithm 1 The CORE($\mathcal{D}$) Protocol.

```
1: procedure Protocol for Agent i_0
2:     upon receiving an external input, do
3:         send "trigger" to all

4: procedure Protocol for Agent i_k
5:     upon receipt of a "trigger" message, do
6:         Wait until current time ≥ D
7:         perform α_k
```

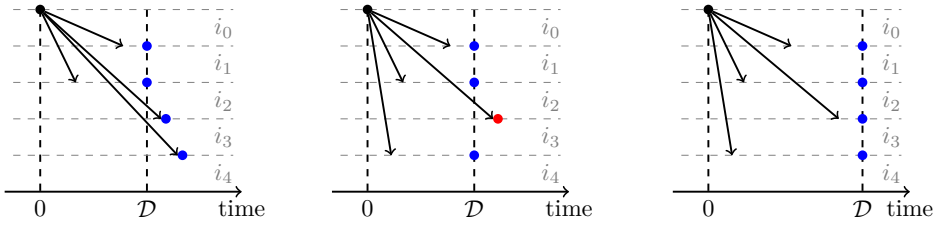
At time $t = 0$, the supervisor agent receives an external input from the environment. The agent then proceeds to broadcast a message to all other agents containing the word "trigger". Agents that receive a "trigger" message from the supervisor, wait until the global clock reads $t = \mathcal{D}$, and then act.[5] If an agent receives the message after time $\mathcal{D}$, it does not wait and acts immediately. Several examples of runs of the CORE protocol are illustrated in Figure 1.

4.2 Probability of a Correct Run

If all messages arrive within time $\mathcal{D}$, then the run will be correct. However, if messages arrive late, this does not mean the run is ruined. For example, suppose the message sent to agent i_k arrives late. Then there is some positive probability that all messages to agents $i_{k+1}, \ldots, i_n$ also arrive late, and that they will arrive in a linear temporal order.

▶ **Theorem 3.** *Fix $\mathcal{D} > 0$, and denote $q \triangleq 1 - e^{-\lambda \mathcal{D}}$, then the probability of a correct run of the CORE($\mathcal{D}$) protocol is $\sum_{k=0}^{n} \frac{1}{k!} q^{n-k} (1-q)^k$.*

[5] Notice that under our protocol several actions can be performed simultaneously, which is consistent with the original definition of Ordered Response in [3]. But the CORE protocol can be easily modified at essentially no cost to support strict ordering of the actions. I.e., if t_i denotes the time at which the i'th action is performed, then $t_{k+1} > t_k$, rather than $t_{k+a} > t_k$, in good runs of the protocol. Namely, for an arbitrarily chosen small ϵ, we can modify line 6 of the protocol so that Agent i_k waits until time $\geq \mathcal{D} + (1 - 2^{-k}) \cdot \epsilon$. In run *(c)* of Figure 1, for example, the actions will be performed in strict linear temporal order, very soon after time $\mathcal{D}$; indeed, they would all complete before time $\mathcal{D} + \epsilon$.



(*a*) A correct run of CORE. (*b*) An incorrect run of CORE. (*c*) Same run, with a larger $\mathcal{D}$.

Figure 1 Three examples of runs of the CORE($\mathcal{D}$) protocol.
At time $t = 0$, agent i_0 broadcasts a message to all other agents. The blue and red dots mark the points in time when the agents perform their actions. (*a*) – Agents i_3 and i_4 receive their messages late and act immediately and in order. This is a lucky run of the protocol. (*b*) – Agents i_3 receives its message late and acts immediately, but out of order. This is an unlucky run of the protocol. (*c*) – Same as the previous run, but $\mathcal{D}$ is chosen to be larger. All agents receive their messages before time $\mathcal{D}$ and act simultaneously.

Proof. In the CORE($\mathcal{D}$) protocol, every agent i_k acts at a time $t_k \geq \mathcal{D}$. Let $e_1, e_2, \ldots, e_n$ be the exponential random variables associated with the messages of the supervisor's broadcast to agents $i_1, i_2, \ldots, i_n$ respectively. for every agent i_k, if $e_k \leq \mathcal{D}$, then $t_k = \mathcal{D}$, and otherwise $t_k = e_k$. Hence, for all $k \geq 1$:

$$t_k = \max\{\mathcal{D}, e_k\}. \tag{1}$$

Notice that the set $\{t_1, t_2, \ldots, t_n\}$ are *i.i.d.* random variables. To calculate the probability $\Pr(t_1 \leq t_2 \leq \cdots \leq t_n)$, we use the Law of Total Probability by conditioning on the values of the set $\{e_k\}_{k=1}^n$ relative to $\mathcal{D}$, i.e:

$$E_0 = e_1 \leq \mathcal{D},\ e_2 \leq \mathcal{D}, \ldots,\ e_{n-2} \leq \mathcal{D},\ e_{n-1} \leq \mathcal{D},\ e_n \leq \mathcal{D} \tag{2}$$
$$E_1 = e_1 \leq \mathcal{D},\ e_2 \leq \mathcal{D}, \ldots,\ e_{n-2} \leq \mathcal{D},\ e_{n-1} \leq \mathcal{D},\ e_n > \mathcal{D} \tag{3}$$
$$E_2 = e_1 \leq \mathcal{D},\ e_2 \leq \mathcal{D}, \ldots,\ e_{n-2} \leq \mathcal{D},\ e_{n-1} > \mathcal{D},\ e_n > \mathcal{D} \tag{4}$$
$$\vdots \tag{5}$$
$$E_n = e_1 > \mathcal{D},\ e_2 > \mathcal{D}, \ldots,\ e_{n-2} > \mathcal{D},\ e_{n-1} > \mathcal{D},\ e_n > \mathcal{D}\,. \tag{6}$$

We use only these $n + 1$ combinations out of the possible 2^n, since for any other event the agents will necessarily act out of order, and the probability of Ordered Response conditioned on such an event would be zero. By the Law of Total Probability:

$$\Pr(t_1 \leq \ldots \leq t_n) = \sum_{k=0}^{n} \Pr(t_1 \leq \ldots \leq t_n \mid E_k) \Pr(E_k). \tag{7}$$

The probability of event E_k is the product of the probability that the first $n - k$ messages are delayed at most $\mathcal{D}$ time, and the probability that the remaining k messages are delayed longer:

$$\Pr(E_k) = q^{n-k}(1 - q)^k. \tag{8}$$

Given an event E_k, it holds that the first $n - k$ agents will act simultaneously. However the remaining k messages arrive late, and the agents act immediately upon their delivery. Due to symmetry, for the last k agents, any temporal ordering of their actions is equally likely. Thus:

$$\Pr(t_1 \leq \ldots \leq t_n \mid E_k) = \Pr(t_{n-k+1} \leq \ldots \leq t_n \mid E_k) = \frac{1}{k!}. \tag{9}$$

Plugging 8 and 9 into Equation 7 yields:

$$\Pr(t_1 \leq \ldots \leq t_n) = \sum_{k=0}^{n} \frac{1}{k!} p^{n-k} (1-p)^k. \tag{10}$$

◀

As can be seen from Theorem 3, the probability of a correct run depends both on $\mathcal{D}$ and n. Also, it includes runs that are "lucky", similarly to the one presented in Figure 1 (a). The probability that a run is correct "by design", i.e., every message arrives by time $\mathcal{D}$, is equal to $q^n = (1 - e^{-\lambda \mathcal{D}})^n$ and, in particular, the probability of a correct run tends to 1 as $\mathcal{D}$ tends to ∞, as can be seen from the following Corollary:

▶ **Corollary 4.** *The probability of a correct run of the CORE($\mathcal{D}$) protocol is at least* $(1 - e^{-\lambda \mathcal{D}})^n$.

Proof. From Theorem 3, the probability of a correct run of CORE($\mathcal{D}$) is given by:

$$\sum_{k=0}^{n} \frac{1}{k!} q^{n-k} (1-q)^k = q^n + \sum_{k=1}^{n} \frac{1}{k!} q^{n-k} (1-q)^k \geq q^n = (1 - e^{-\lambda \mathcal{D}})^n. \tag{11}$$

◀

Corollary 4 characterizes the relationship between the probabilistic communication bound $\mathcal{D}$ and the probability of a correct run of the CORE protocol. We now derive a closed-form expression for the probabilistic communication bound $\mathcal{D}$.

▶ **Corollary 5.** *Fix $0 < p < 1$, and let $\mathcal{D} \geq \frac{-\ln(1-\sqrt[n]{p})}{\lambda}$. Then the probability of a correct run of the CORE($\mathcal{D}$) protocol in a setting with a global clock, is at least p.*

Proof. Follows directly from Corollary 4: $\mathcal{D} \geq \frac{-\ln(1-\sqrt[n]{p})}{\lambda} \Rightarrow p \leq (1 - e^{-\lambda \mathcal{D}})^n$. ◀

4.3 Expected Response Time

We are interested in the expected response time of the CORE protocol. Specifically, this is the expected time of the last agent to perform its action. In a given run of the CORE protocol, the response time is determined by the arrival times of the messages from the supervisor's broadcast. They either all arrive by time $\mathcal{D}$, or at least one of them is delayed longer.

Let $e_1, e_2, \ldots, e_n$ be the set of exponential random variables that are associated with messages sent in the supervisor's broadcast. Let $\mathcal{M} = \max\{e_1, e_2, \ldots, e_n\}$ be a random variable that denotes the time when the last message in a run is delivered. Then the response time of a run of the protocol is given by $RT = \max\{\mathcal{D}, \mathcal{M}\}$.

We are interested in the expected value of the response time $\mathbb{E}[RT]$. However, deriving the expectation of the above non-linear function of a random variable and a constant is a non-trivial exercise. Fortunately, there is a simple bound on $\mathbb{E}[RT]$ that is good enough for our purposes:

$$\mathbb{E}[RT] = \mathbb{E}[\max\{\mathcal{D}, \mathcal{M}\}] \leq \mathcal{D} + \mathbb{E}[\mathcal{M}]. \tag{12}$$

The inequality follows from the fact the maximum is smaller than the sum, and the linearity of probabilistic expectation. Before we proceed in characterizing the expected response time, we prove a useful lemma:

▶ **Lemma 6.** *$E[\mathcal{M}] = \frac{H_n}{\lambda}$, where H_n is the n^{th} harmonic number; i.e. $H_n = \sum_{m=1}^{n} \frac{1}{m}$.*

Proof. Recall that $\mathcal{M} = \max\{e_1, e_2, \ldots, e_n\}$. The random variables $\{e_k\}_{k=1}^{n}$ are assumed to be *i.i.d.* exponential random variables with parameter λ. Denote by T_m to be the m^{th} smallest of the set $\{e_k\}_{k=1}^{n}$, i.e., $T_1 = \min_k\{e_k\}$ and $T_m = \min\left\{\{e_k\}_{k=1}^{n} \setminus \{T_1, \ldots, T_{m-1}\}\right\}$.

It is well known that the minimum of n *i.i.d.* exponential random variables with parameter λ is also an exponential random variable with parameter $n\lambda$. Thus, $\mathbb{E}[T_1] = \frac{1}{n\lambda}$.

Since exponential random variables are continuous, the probability that any two of the n random variables have the same value is 0. Thus, the $n-1$ other random variables all have values strictly larger than T_1. However, the memorylessness property of the exponential distribution implies that knowledge of T_1 essentially "resets" the values of the other random variables, so that the time between T_1 and T_2 is the same (distributionally) as the time until the first of $n-1$ *i.i.d.* exponential random variables takes on a value. Hence, $\mathbb{E}[T_2 - T_1] = \frac{1}{(n-1)\lambda}$.

By inductive reasoning, we get that $\forall 1 \leq m \leq n-1 : \mathbb{E}[T_{m+1} - T_m] = \frac{1}{(n-m)\lambda}$. As a result, the expected value of the maximum of the n exponential random variables is:

$$\mathbb{E}[\mathcal{M}] = \mathbb{E}[T_n] = \mathbb{E}\left[T_1 + \sum_{m=1}^{n-1}(T_{m+1} - T_m)\right] = \sum_{m=0}^{n-1}\frac{1}{(n-m)\lambda} = \sum_{m=1}^{n}\frac{1}{m\lambda} = \frac{H_n}{\lambda}. \tag{13}$$

◀

The harmonic numbers roughly approximate the natural logarithm function [7], i.e., $H_n \in \Theta(log(n))$. As a result, we gain the following theorem:

▶ **Theorem 7.** *Fix $0 < p < 1$, and let $\mathcal{D} \geq \frac{-\ln(1-\sqrt[n]{p})}{\lambda}$. The expected response time of the CORE($\mathcal{D}$) protocol is logarithmic in the number of participating agents i.e., $\mathbb{E}[RT] \in \mathcal{O}\left(\frac{\log(n)}{\lambda}\right)$.*

Proof. From Inequality 12:

$$\mathbb{E}[RT] \leq \mathcal{D} + \mathbb{E}[\mathcal{M}]. \tag{14}$$

We plug in the expressions for $\mathcal{D}$ and $\mathbb{E}[\mathcal{M}]$ from Corollary 5 and Lemma 6:

$$\mathbb{E}[RT] \leq \mathcal{D} + \mathbb{E}[\mathcal{M}] \ = \frac{1}{\lambda} \cdot \left(-\ln(1 - \sqrt[n]{p}) + H_n\right). \tag{15}$$

Notice that $-\ln(1 - \sqrt[n]{p}) \in \Theta(\log(n))$ for any $0 < p < 1$:

$$\lim_{n\to\infty} \frac{-ln(1-\sqrt[n]{p})}{\ln(n)} \underset{\text{Heine}}{=} \lim_{x\to\infty} \frac{-ln(1-\sqrt[x]{p})}{\ln(x)} \underset{\text{L'Hôpital}}{=} \lim_{x\to\infty} \frac{-\frac{1}{1-\sqrt[x]{p}} \cdot \frac{\sqrt[x]{p}\ln(p)}{x^2}}{\frac{1}{x}} \tag{16}$$

$$= \lim_{x\to\infty} \frac{-\frac{\ln(p)}{x}}{\frac{1-\sqrt[x]{p}}{\sqrt[x]{p}}} = \lim_{x\to\infty} \frac{-\frac{\ln(p)}{x}}{-\sqrt[x]{p}-1} \underset{\text{L'Hôpital}}{=} \lim_{x\to\infty} \frac{\frac{\ln(p)}{x^2}}{\frac{-\sqrt[x]{p}\ln(p)}{x^2}} \tag{17}$$

$$= \lim_{x\to\infty} \sqrt[x]{p} = 1. \tag{18}$$

Additionally, the harmonic numbers roughly approximate the natural logarithm function [7], i.e., $H_n \in \Theta(log(n))$. Thus, $\mathbb{E}[RT] \in \mathcal{O}\left(\frac{\log(n)}{\lambda}\right)$. ◀

4.4 The CORE-Message-Chain Hybrid Protocol

From Theorem 7, we see that the CORE protocol's expected response time grows logarithmically in n. In comparison, the Message Chain protocol's expected response time grows linearly in n. Hence, the CORE protocol is exponentially faster than the Message Chain protocol. However, the Message Chain protocol, while slower for large n, guarantees Ordered Response *w.p.* 1. Nonetheless, a case can be made in favour of the CORE protocol if it can maintain its edge for any n, and for any probability of a correct run p that is as close to 1 as desirable.

Currently, this is not the case. Recall that $\mathcal{D}$ tends to ∞ as p tends to 1. As a result, for any given n, there exists some probability $\bar{p}$ such that for all $\bar{p} < p < 1$ the Message Chain protocol's expected response time $\frac{n}{\lambda}$ is smaller than $\mathcal{D} = \frac{-\ln(1-\sqrt[n]{p})}{\lambda}$. The key insight here is that for small n the Message Chain protocol is very efficient and quick, but for large n it suffers from a significant slowdown in performance. Our solution is to combine the two protocols. Essentially, we now design a protocol that runs both protocols concurrently, in order to benefit from the best of both worlds.

In this hybrid protocol, the supervisor broadcasts the "trigger" message to all agents, and also sends a special message containing the word "act" to i_1. When i_1 receives this message, it starts a message chain that will pass through all agents. Agents that receive a message chain perform their action immediately, pass it on to the next agent in the ordering, and then terminate. Until then, they behave according to the CORE protocol described in Algorithm 1.

4.5 Performance and Probability of Correctness

Deriving an exact expression for the probability of a correct run in the hybrid protocol is difficult. This is mainly due to the fact that in the hybrid protocol an agent may perform its action for two reasons. The agent may act either because it received a message chain, or because it previously received a message from the supervisor and the the global time is $\mathcal{D}$. It is enough for our purposes to lower bound the probability of a correct run. Similarly to the pure CORE protocol, if all messages from the trigger broadcast are delivered by time $\mathcal{D}$, then Ordered Response is guaranteed. From Corollary 4 we have that:

▶ **Theorem 8.** *Fix a parameter $\mathcal{D} > 0$. The probability of a correct run in the CORE($\mathcal{D}$)-Message-Chain protocol is at least $(1 - e^{\lambda \mathcal{D}})^n$.*

We focus the reader's attention on two important observations. The first is that, in terms of expected response time, the hybrid protocol is no worse than the Message Chain protocol in every run. Once an agent receives a message chain, the agent immediately performs its action and terminates. The second observation is that, from some value of n, the expected response time stops growing linearly in n, and starts growing logarithmically in n. Thus, it holds for the hybrid protocol that $\mathbb{E}[RT] \in \mathcal{O}\left(\frac{\log(n)}{\lambda}\right)$.

With high probability, the message chain is automatically truncated once time $\mathcal{D}$ is reached, since no agent propagates the message chain after termination. For small n, and large $\mathcal{D}$, the message chain will more likely terminate the agents before time $\mathcal{D}$, which yields performance linear in n. However, when n is large, the message chain will be much slower. In this case, the CORE part of the hybrid protocol dominates, and the expected response time will be logarithmic in n.

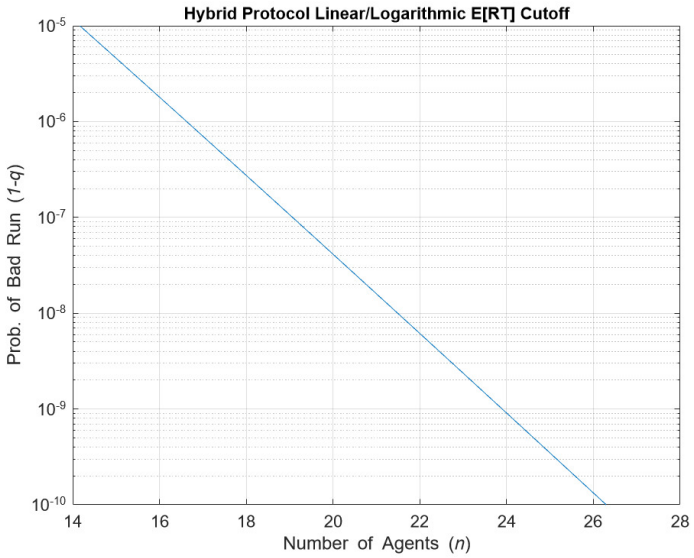


Figure 2 Plot of the Linear/Logarithmic expected response time of the CORE-Message-Chain protocol.

In Figure 2, given some probability of an incorrect run $1 - p$, we plot for which n it holds that $\frac{n}{\lambda} = \mathcal{D}$. The plot illustrates the linear/logarithmic cutoff point of the hybrid protocol, i.e., for a given probability of a correct run $p < 1$, what is the minimum number of participating agents N required such that for all $n > N$ the expected response time stops growing linearly in n, and starts growing logarithmically in n.

Figure 2 shows that the hybrid protocol exhibits linear performance only when n is small. For example, if we allow the rate of incorrect runs to be 1 in 10^6 (i.e., $p = 1 - 10^{-6}$), then a lower bound on the number of participating agents needed for the hybrid protocol to exhibit logarithmic performance is roughly $n = 18$. For $p = 1 - 10^{-9}$, the lower bound is $n = 24$. As can be seen from the slope of the graph, the hybrid protocol requires an additional 3 participating agents for a tenfold improvement in the probability of correctness. Hence, correctness probability of $p = 1 - 10^{-100}$, would require roughly $n = 300$ agents.

5 CORE without a Global Clock

In Section 4, we presented a concurrent Ordered Response protocol for the EDD model when we can assume the existence of an accurate global clock. However, agents are only assumed to be able to measure time accurately using their own local clocks, though they may be initially offset relative to each other by some unknown quantity.

We now consider the case where agents do not share a global clock, and so they initially have no idea how early or late they are relative to other agents. Previously, we assumed that the environment's external input would arrive at a point in time that we could denote as "$t = 0$", and that when an agent becomes active it would have access to the current global time. However, in the current model, the arrival of the initial external input is only observed by the supervisor, and the knowledge of when it arrived cannot be disseminated accurately to all the other agents.

This is a major challenge to the design of a concurrent Ordered Response protocol. To overcome this, we again leverage the probability distribution on message delays, to *probably approximately* synchronize the local clocks of the agents. Specifically, the worker agents will attempt to approximate when the external input has arrived based on the arrival times of messages broadcast by the supervisor and synchronize their local clocks to fit this hypothesis.

5.1 Probably Approximately Synchronized Local Clocks

For ease of exposition, from this point onward assume that in addition to the supervisor agent i_0, there are $n+1$ worker agents named $i_1, i_2, \ldots, i_{n+1}$. As before, the supervisor agent i_0 broadcasts a "trigger" message to all the other agents. When a worker agent receives a trigger message it rebroadcasts it to all of its coworkers, with the message "redirect". Concurrently, it waits for n such "redirect" messages, and logs a timestamp of the arrival time of each such message according to its local clock. When the final message arrives, the agent proceeds to calculate its hypothesis of when the external input arrived based on the observed timestamps.

Agent i_k's hypothesis T_k is the mean of the measured timestamps $\{\tau_1, \tau_2, \ldots, \tau_n\}$ minus the expected delay of a message chain of length 2, i.e. $T_k = \frac{1}{n}\sum_{m=1}^{n}\tau_m - \frac{2}{\lambda}$. Notice that the sequence of timestamps are *i.i.d.* Erlang[6] random variables with parameters 2 and λ. By the Law of Large Numbers, as $n \to \infty$ their mean converges to their expected value: $\frac{2}{\lambda}$. This means that $T_k \xrightarrow{n\to\infty} 0$, or, in this case, it converges to the true arrival time of the external input to the system.

The value T_k is used by agent i_k to offset its local clock. Formally, we denote by $C_{i_k}(t)$ the time that the local clock of agent i_k shows at time t. We assume that $C_{i_0}(t) = t$, and that the external input arrives at $t = 0$. When an agent i_k hypothesizes that the external input arrived at time T_k, it sets an adjusted local clock $C_{i_k}^{Adj}(t)$ to be the value of $C_{i_k}(t)$ offset by T_k. I.e., for all $t > 0$, agent i_k's adjusted local clock shows that $C_{i_k}^{Adj}(t) = C_{i_k}(t) - T_k$.

Since the number of agents in the system is finite, the agents' local clocks are still offset relative to the supervisor's local clock. However, when the aforementioned procedure concludes, the relative offset between agents' local clocks can be bounded with some probability.

▶ **Theorem 9.** *Let $\tau_m \sim Erlang(2,\lambda)$ for $m = 1,\ldots,n$, and let $T_k = \frac{1}{n}\sum_{m=1}^{n}\tau_m - \frac{2}{\lambda}$. Then:*

$$\Pr\left(\max_{k\geq 1}|T_k| \leq \frac{\ln(n)}{\lambda\sqrt{n}}\right) \geq \Psi(n) \tag{19}$$

where:

$$\Psi(n) \triangleq \max\left\{0,\ 1 - \left(1 - \frac{\ln(n)}{2\sqrt{n}}\right)^{2n} n^{1+\sqrt{n}} - \left(1 + \frac{\ln(n)}{2\sqrt{n}}\right)^{2n} n^{1-\sqrt{n}}\right\}. \tag{20}$$

In particular, $\Psi(n) \xrightarrow{n\to\infty} 1$.

The proof of Theorem 9 is quite lengthy and appears in the extended version of this paper [16] in Appendix B.4. Denote $\delta \triangleq \frac{\ln(n)}{\lambda\sqrt{n}}$. Then by Theorem 9, with probability at least $\Psi(n)$, the local clocks of all agents are δ-*synchronized*:

▶ **Definition 10.** *If for all $k, m \geq 1$ it holds that $\left|C_{i_k}^{Adj}(t) - C_{i_m}^{Adj}(t)\right| \leq \delta$, we say that the agents' local clocks are* δ-synchronized.

[6] An Erlang random variable is the probabilistic distribution of a sum of *i.i.d.* exponential random variables (see [9]).

As can be seen from Theorem 9, the probability that the agents' local clocks will be δ-synchronized converges to 1 as $n \to \infty$. However, for $n < 115$, $\Psi(n) = 0$. This is due to the fact that in the proof of Theorem 9, we use the Union Bound Theorem, which results in an overly course lower bound. As a result, for these values of n, the probabilistic bound given in Theorem 9 is not very informative. To see the actual probability of δ-synchronization for small n, we have performed a small experiment. We sampled the vector $(T_1, T_2, \ldots, T_n)$ a thousand times and calculated the empirical mean over the binary results of whether the vector elements are at most 2δ far from each other. As can be seen in Figure 3, the real probability of δ-synchronization is much higher, and for $n > 400$ is nearly 1.

5.2 The PA-CORE Protocol

In a setting without a global clock, we design a two phase protocol that we call PA-CORE for *Probable Approximate* Concurrent Ordered Response. The first phase is the δ-synchronization of the agents' local clocks using the procedure described in Section 5.1. In the second phase, the agents use this to perform their actions in a linear temporal order. The pusdeocode of the protocol is presented in Algorithm 2.

Algorithm 2 The PA-CORE($\mathcal{D}$) Protocol.

procedure PROTOCOL FOR AGENT i_0
  upon receiving an external input, do
    send "**trigger**" to all

procedure PROTOCOL FOR AGENT i_k
  Setup:
    $m \leftarrow 0$

  upon receipt of a "trigger" message, do
    send "**redirect**" to all

  upon receipt of a "redirect" message, do
    $m \leftarrow m + 1$
    $\tau_m \leftarrow C_{i_k}(t)$ ▷ Measure and record current time.
    if $m = n$ **then**
      $T \leftarrow \frac{1}{n}\sum_{m=1}^{n} \tau_m - \frac{2}{\lambda}$
      $C_{i_k}^{Adj}(t) \leftarrow C_{i_k}(t) - T$ ▷ Set adjusted local clock.
      Wait until $C_{i_k}^{Adj}(t) \geq \mathcal{D} + 2\delta \cdot (k-1)$
      perform α_k

At time $t = 0$, agent i_0 broadcasts a "trigger" message to all other agents. When a worker agent i_k receives a "trigger" message, it broadcasts a message with the word "redirect" to all worker agents. Upon receiving a "redirect" message, the agent records a timestamp according to its local clock. After n such "redirect" messages have been received, the agent approximates the arrival time of the external input and offsets its local clock to match this hypothesis.

Now, the agent i_k moves into phase 2. Being the k-th agent in the order, the agent waits until time $\mathcal{D} + 2\delta \cdot (k-1)$, and then performs its action. If the local agent's time is already past this point when it receives the last "redirect" message, it performs its action immediately. The additional delay of $2\delta \cdot (k-1)$ for agent i_k ensures that agents act in non-overlapping time windows.

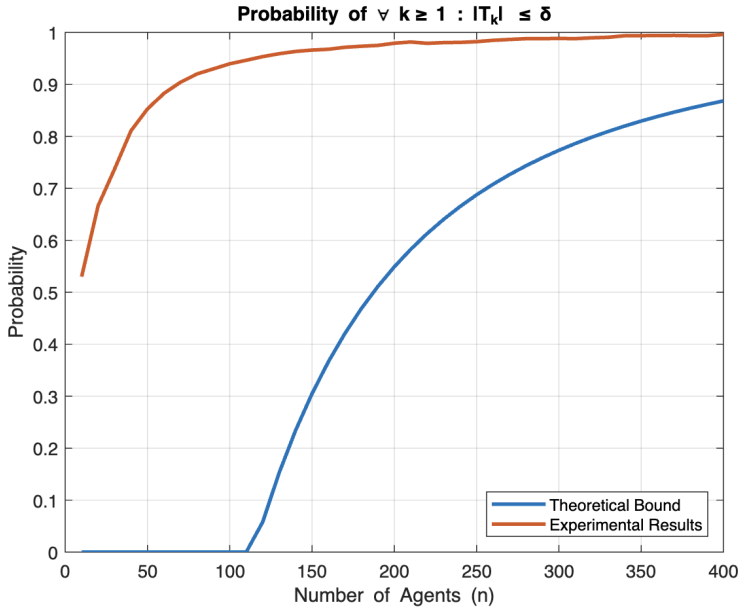


Figure 3 Probability that clocks are δ-synchronized.

Notice that if n was infinite, then $\delta = 0$, and the PA-CORE protocol would closely mirror the CORE protocol in Section 4. Intuitively, the first phase of the protocol attempts to δ-synchronize the agents' local clocks with high probability. However, δ-synchronization is not a global clock, and so agents cannot act simultaneously. Instead, in phase 2 of the protocol, the agents wait an index-dependent time delay that guarantees that they act in non-overlapping windows.

In contrast to the Message Chain protocol, the expected delay between the acting times of two consecutive agents is not a constant. In PA-CORE, this delay is at most 2δ *w.h.p.* (assuming that the agents' local clocks are δ-synchronized). Recall that $\delta \triangleq \frac{\ln(n)}{\lambda\sqrt{n}}$. This value was chosen because the product $n\delta$ is in $\mathcal{O}\left(\frac{\sqrt{n}\log(n)}{\lambda}\right)$ which we will show results in the expected response time of the protocol to grow sub-linearly in n.

5.3 Probability of a Correct Run

▶ **Lemma 11.** *In a run of the PA-CORE($\mathcal{D}$) protocol, if all of the messages arrive by time $\mathcal{D}$ and the agents' local clocks are δ-synchronized, then the run is guaranteed to be correct.*

Proof. We prove the claim by induction on the number of worker agents $n+1$.

Base Case: ($n = 1$) Since all messages arrive by time $\mathcal{D}$, and the agents' local clocks are δ-synchronized, we have that $C_{i_1}^{Adj}(\mathcal{D}) \leq \mathcal{D}+\delta$. Therefore, agent i_1 acts no later than time $\mathcal{D}+\delta$. Also, for agent i_2 we have that $C_{i_2}^{Adj}(\mathcal{D}) \geq \mathcal{D}-\delta$. Since agent i_2 waits an additional 2δ before acting, agent i_2 acts no earlier than time $\mathcal{D}+\delta$. Thus, agents i_1 and i_2 act in the proper linear temporal order. Step: Suppose that the claim holds for all $k \leq n$. By the same logic as in the base case, agent i_{n+1} acts only after agent i_n, and by the induction hypothesis, agent i_n also acts only after all the previous agents in the ordering have acted. ◀

▶ **Theorem 12.** *Fix $\mathcal{D} > 0$. The probability of a correct run of the PA-CORE($\mathcal{D}$) protocol is at least $\Psi(n) \cdot (1 - e^{-0.5\lambda\mathcal{D}})^{(n+1)^2}$.*

Proof. By Lemma 11, if all messages arrive by time $\mathcal{D}$ and all the agents' local clocks are δ-synchronized, then Ordered Response is assured. Therefore, the probability of a correct run of the protocol is greater or equal to the probability that both of these events occur.

We now derive the probability that all the messages arrive by time $\mathcal{D}$. For agent i_k, let e_k be the exponential random variable associated with the delivery time of the message sent from i_0 to i_k in agent i_0's initial broadcast. Let M_k be the maximum of the n random variables that are associated with agent i_k's redirect messages. Then, the probability that all messages arrive by time $\mathcal{D}$ is:

$$\Pr\left(\max_{1\leq k\leq n+1}\{e_k + M_k\} \leq \mathcal{D}\right) \geq \Pr\left(\max_{1\leq k\leq n+1}\{2\cdot\max\{e_k, M_k\}\} \leq \mathcal{D}\right) \tag{21}$$

$$= \Pr\left(\max_{1\leq k\leq n+1}\{\max\{e_k, M_k\}\} \leq 0.5\,\mathcal{D}\right) \tag{22}$$

where the inequality above follows from the fact that $e_k + M_k \leq 2\cdot\max\{e_k, M_k\}$. Notice that for all $k \geq 1$ the random variables in the set $\{\max\{e_k, M_k\}\}_{k\geq 1}$ are independent of one another. Hence:

$$\Pr\left(\max_{1\leq k\leq n+1}\{\max\{e_k, M_k\}\} \leq 0.5\,\mathcal{D}\right) = \Pr\left(\{\max\{e_k, M_k\}\} \leq 0.5\,\mathcal{D}\right)^{n+1}. \tag{23}$$

Notice that $\max\{e_k, M_k\}$ is the maximum over $n+1$ *i.i.d.* exponential random variables, and therefore the CDF of the maximum is the product of their individual CDFs:

$$\Pr\left(\{\max\{e_k, M_k\}\} \leq 0.5\,\mathcal{D}\right) = (1 - e^{-0.5\lambda\,\mathcal{D}})^{n+1}. \tag{24}$$

Hence, the probability that all of the messages arrive by time $\mathcal{D}$ is:

$$\tilde{p} = (1 - e^{-0.5\lambda\,\mathcal{D}})^{(n+1)^2}. \tag{25}$$

Let $E_{\mathcal{D}}$ denote the event that all messages arrive by time $\mathcal{D}$, and let E_δ denote the event that all the agents' local clocks are δ-synchronized. By Lemma 11 and Theorem 9, we obtain:

$$p \geq \Pr(E_\delta \wedge E_{\mathcal{D}}) = \Pr(E_\delta | E_{\mathcal{D}})\Pr(E_{\mathcal{D}}) \geq \Psi(n)\cdot(1 - e^{-0.5\lambda\,\mathcal{D}})^{(n+1)^2}. \tag{26}$$

◀

Denote by $\tilde{p}$ the probability that all messages arrive by time $\mathcal{D}$. By Theorem 12, $\tilde{p} = (1 - e^{-0.5\lambda\,\mathcal{D}})^{(n+1)^2}$, and the probability of a correct run is lower bounded by $\Psi(n)\cdot\tilde{p}$, where $\Psi(n)$ is as defined in Theorem 9. The probability $\tilde{p}$ depends on the value of $\mathcal{D}$, and as $\mathcal{D}$ is increased, $\tilde{p}$ draws closer to 1.

The lower bound we have derived is a product of two probabilities, $\Psi(n)$ and $\tilde{p}$. While the latter is controllable by $\mathcal{D}$, the former is static, since it is solely dependent on n. From this point onward, we focus our analysis for large n, and we assume that $\Psi(n) \approx 1$, as the experimental results that are illustrated in Figure 3, suggest. We leave the proof of a tighter lower bound for future work.

▶ **Corollary 13.** *Fix $0 < p < 1$, and let $\mathcal{D} \geq \frac{-2\ln(1-\sqrt[(n+1)^2]{p})}{\lambda}$. Then, the probability of a correct run of the PA-CORE($\mathcal{D}$) protocol is at least p.*

Proof. From Theorem 12, the probability of a correct run is lower bounded by $(1 - e^{-0.5\lambda\,\mathcal{D}})^{(n+1)^2}$. Thus: $\mathcal{D} \geq -\frac{2\ln\left(1-\sqrt[(n+1)^2]{p}\right)}{\lambda} \Rightarrow p \leq (1 - e^{-0.5\lambda\,\mathcal{D}})^{(n+1)^2}$. ◀

5.4 Expected Response Times

Let RT_k be the response time of agent i_k in a run of the protocol. Before an agent performs its action, it waits for n message chains of length 2, which it uses to approximate the external input arrival time. Independently of that, the agent also relays a message from the supervisor's original broadcast. Hence, the agent's response time is the maximum over two random variables.

Formally, let e_k be the exponential random variables associated with the delivery time of the "trigger" message sent to agent i_k, and let $\tau_1^k, \tau_2^k, \ldots, \tau_n^k$ be the timestamps measured by the agent when its receives the "redirect" messages. Notice that $\{\tau_m^k\}_{m=1}^n$ is a set of *i.i.d.* Erlang random variables with parameters 2 and λ. Let $\mathcal{T}^k \triangleq \max\{\tau_1^k, \tau_2^k, \ldots, \tau_n^k\}$, then:

$$RT_k = \max\left\{e_k, \max\left\{\mathcal{T}^k, \mathcal{D} + 2\delta \cdot (k-1)\right\}\right\} \tag{27}$$

$$= \max\left\{e_k, \mathcal{T}^k, \mathcal{D} + 2\delta \cdot (k-1)\right\} \leq \max\left\{e_k, \mathcal{T}^k, \mathcal{D} + 2\delta n\right\}. \tag{28}$$

Then, the expected response time can be upper bounded:

$$RT = \max_k \{RT_k\} = \max_k \left\{\max\left\{e_k, \mathcal{T}^k, \mathcal{D} + 2\delta n\right\}\right\} = \max\left\{\max_k \left\{e_k, \mathcal{T}^k\right\}, \mathcal{D} + 2\delta n\right\} \tag{29}$$

$$\leq \max_k \left\{e_k, \mathcal{T}^k\right\} + \mathcal{D} + 2\delta n \quad \Rightarrow \quad \mathbb{E}[RT] \leq \mathbb{E}\left[\max_k \left\{e_k, \mathcal{T}^k\right\}\right] + \mathcal{D} + 2\delta n. \tag{30}$$

In order to prove that the expected response time of the PA-CORE($\mathcal{D}$) protocol is sublinear in the number of agents, we first prove the following Lemma:

▶ **Lemma 14.** *The following holds:* $\mathbb{E}\left[\max_k \left\{e_k, \mathcal{T}^k\right\}\right] \leq \frac{2H_{n^2+n}}{\lambda}$.

Proof. Recall that $\mathcal{T}^k = \max\left\{\tau_1^k, \tau_2^k, \ldots, \tau_n^k\right\}$. Each timestamp τ_m^k is measured when the agent i_k receives a "redirect" message, which completes a message chain composed of two messages. Hence, τ_m^k is a sum of two exponential random variables, i.e., $\tau_m^k = e_{m,1}^k + e_{m,2}^k$. Notice that $e_{m,1}^k + e_{m,2}^k \leq 2\max\left\{e_{m,1}^k, e_{m,2}^k\right\}$. We have that:

$$\mathcal{T}^k = \max\left\{\tau_1^k, \tau_2^k, \ldots, \tau_n^k\right\} \leq \max_{1\leq m\leq n}\left\{2\max\left\{e_{m,1}^k, e_{m,2}^k\right\}\right\} = 2\max_{\substack{1\leq m\leq n\\ \ell\in\{1,2\}}}\left\{e_{m,\ell}^k\right\}$$

$$\Rightarrow \quad \mathbb{E}\left[\max_k \left\{e_k, \mathcal{T}^k\right\}\right] \leq \mathbb{E}\left[\max_k\left\{e_k, 2\max_{\substack{1\leq m\leq n\\ \ell\in\{1,2\}}}\left\{e_{m,\ell}^k\right\}\right\}\right]$$

$$\leq \mathbb{E}\left[\max_k\left\{2e_k, 2\max_{\substack{1\leq m\leq n\\ \ell\in\{1,2\}}}\left\{e_{m,\ell}^k\right\}\right\}\right]$$

$$= 2\,\mathbb{E}\left[\max_k\left\{e_k, \max_{\substack{1\leq m\leq n\\ \ell\in\{1,2\}}}\left\{e_{m,\ell}^k\right\}\right\}\right].$$

Notice that the argument inside the expectation operator is just a maximum over $n^2 + n$ *i.i.d.* exponential random variables. By Lemma 6, we have that:

$$\mathbb{E}\left[\max_k \left\{e_k, \mathcal{T}^k\right\}\right] \leq 2\,\mathbb{E}\left[\max_k\left\{e_k, \max_{\substack{1\leq m\leq n\\ \ell\in\{1,2\}}}\left\{e_{m,\ell}^k\right\}\right\}\right] \leq \frac{2H_{n^2+n}}{\lambda}. \tag{31}$$

◀

▶ **Theorem 15.** *The expected response time of the PA-CORE($\mathcal{D}$) protocol is sub-linear in the number of participating agents. In particular,* $\mathbb{E}[RT] \in \mathcal{O}\left(\frac{\sqrt{n}\log(n)}{\lambda}\right)$.

Proof. By Lemma 14:

$$\mathbb{E}[RT] \quad \leq \quad \frac{2H_{n^2+n}}{\lambda} + \mathcal{D} + 2\delta n. \tag{32}$$

We now that prove each expression in the sum is sub-linear in n:

- For $\frac{2H_{n^2+n}}{\lambda}$:

$$H_{n^2+n} \approx \ln(n^2+n) \in \mathcal{O}(\log(n)) \quad \Rightarrow \quad \frac{2H_{n^2+n}}{\lambda} \in \mathcal{O}\left(\frac{\log(n)}{\lambda}\right) \tag{33}$$

- For $\mathcal{D}$:
  By Corollary 13:

$$\mathcal{D} \approx -\frac{2\ln\left(1 - \sqrt[(n+1)^2]{p}\right)}{\lambda}. \tag{34}$$

  Notice that $-\ln(1 - \sqrt[(n+1)^2]{p}) \in \Theta(\log(n))$ for any $0 < p < 1$:

$$\lim_{n\to\infty} \frac{-ln(1 - \sqrt[(n+1)^2]{p})}{\ln(n+1)} \underset{\text{Heine}}{=} \lim_{x\to\infty} \frac{-ln(1 - \sqrt[(x+1)^2]{p})}{\ln(x+1)} \tag{35}$$

$$\underset{\text{L'Hôpital}}{=} \lim_{x\to\infty} \frac{-\frac{1}{1-\sqrt[(x+1)^2]{p}} \cdot \frac{2\cdot\sqrt[(x+1)^2]{p}\ln(p)}{(x+1)^3}}{\frac{1}{x+1}} \tag{36}$$

$$= \lim_{x\to\infty} \frac{-\frac{2\ln(p)}{(x+1)^2}}{\frac{1-\sqrt[(x+1)^2]{p}}{\sqrt[(x+1)^2]{p}}} = \lim_{x\to\infty} \frac{-\frac{2\ln(p)}{(x+1)^2}}{-\sqrt[(x+1)^2]{p} - 1} \tag{37}$$

$$\underset{\text{L'Hôpital}}{=} \lim_{x\to\infty} \frac{\frac{4\ln(p)}{(x+1)^3}}{\frac{2\cdot\sqrt[(x+1)^2]{p}\ln(p)}{(x+1)^3}} = \lim_{x\to\infty} 2 \cdot \sqrt[(x+1)^2]{p} = 2. \tag{38}$$

- For $2\delta n$:
  By definition, $\delta = \frac{\ln(n)}{\lambda\sqrt{n}}$. Hence:

$$2\delta n = \frac{2\sqrt{n}\ln(n)}{\lambda} \in \mathcal{O}\left(\frac{\sqrt{n}\log(n)}{\lambda}\right). \qquad \blacktriangleleft$$

In conclusion, we have presented a concurrent Ordered Response protocol that guarantees a correct run with high probability in a sub-linear expected response time. However, the trade-off is increased message complexity. Due to the "redirect" broadcast that every worker agent performs, we have that the message complexity of the PA-CORE protocol is $\Theta(n)$ broadcasts.

5.5 The PA-CORE-Message-Chain Hybrid Protocol

The PA-CORE protocol guarantees *Ordered Response* with high probability in sub-linear expected response time, when $\Psi(n) \approx 1$. We know that $\Psi(n) \approx 1$ when the number of participating agents is very large. Consequently, we have not given any guarantees on the performance of the PA-CORE protocol for small n.

Similarly to the approach we have presented in the EDD model with a global clock, we propose concurrently running the PA-CORE and Message Chain protocols in EDD model without a global clock. The advantages of doing so are two-fold. Firstly, as before, the expected response time of the protocol will be no worse than that of the Message Chain protocol's, and for large n, the expected response time will be sub-linear in n. Secondly, our analysis for the expected response time and probability of a correct run are only accurate and informative for large n.

With the addition of a concurrent message chain, the performance of the hybrid protocol is well-known to us for all values of n. For small n, the message chain guarantees quick *Ordered Response w.p.* 1. For large n, the PA-CORE part of the hybrid protocol dominates, and guarantees *Ordered Response* with high probability in sub-linear expected time. As a result, the hybrid PA-CORE-Message-Chain protocol in a setting without a global clock, guarantees *Ordered Response* with high probability in sub-linear expected time.

6 Discussion and Conclusions

One of the goals of the theory of distributed systems is to understand how various properties of the system affect the solvability and efficiency of distributed problems. To this end, the impact that communicating over channels with probabilistic guarantees on message delivery times has on the efficiency of distributed protocols has received little attention in our community. A comprehensive study of the impact of probabilistic channels on coordination could ultimately involve a taxonomy of different probabilistic assumptions, a variety of coordination problems, tight upper and lower bounds, and an assessment of practical use cases and practical implementation details. This, of course, is a grand research program that goes well beyond the scope of a single conference paper. Our purpose in this paper is to take several initial steps in this direction. To this end, clarity and simplicity are central. Indeed, our goal is to illustrate that such questions can be formulated, and that probabilistic channels can provide a genuine advantage. As we have discussed, asynchronous protocols should be directly implementable in settings in which transmission over a probabilistic channel is guaranteed to succeed eventually with probability 1. In the setting of a natural coordination problem, Ordered Response, and a popular and mathematically accessible exponential distribution over transmission times, we showed is possible to improve over the asynchronous solution in a significant manner. This is true both if clocks are synchronized and when they are not. The solutions that we provide are novel, and their analysis is rigorous. Their simplicity is a feature, and not a bug. Being an initial foray into the topic, our treatment opens the door for many extensions, improvements and refinements, left for future research. Proving lower bounds and matching upper bounds is one. Exploring other probabilistic distributions, and possibly other coordination problems is another. Finally, we consider modifying the treatment for concrete practical settings an important open problem. Much is left to explore regarding coordination in this class of models. By establishing that the complexity of asynchronous solutions can be significantly improved on, we have provided strong motivation for such investigations.

References

1 Hisham S Abdel-Ghaffar. Analysis of synchronization algorithms with time-out control over networks with exponentially symmetric delays. *IEEE Transactions on Communications*, 50(10):1652–1661, 2002. doi:10.1109/TCOMM.2002.803979.

2 Koshal Arvind. Probabilistic clock synchronization in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):474–487, 1994. doi:10.1109/71.282558.

3 Ido Ben-Zvi and Yoram Moses. Beyond lamport's happened-before: On the role of time bounds in synchronous systems. In *International Symposium on Distributed Computing*, pages 421–436. Springer, 2010. doi:10.1007/978-3-642-15763-9_42.

4 Nicolas Bourbaki. *General Topology: Chapters 1–4*, volume 18. Springer Science & Business Media, 2013. doi:10.1007/978-3-642-61701-0.

5 K. Mani Chandy and Jayadev Misra. How processes learn. In *Proceedings of the fourth annual ACM symposium on Principles of Distributed Computing*, pages 204–214, 1985. doi:10.1145/323596.323615.

6 Qasim M Chaudhari, Erchin Serpedin, and Jang-Sub Kim. Energy-efficient estimation of clock offset for inactive nodes in wireless sensor networks. *IEEE Transactions on Information Theory*, 56(1):582–596, 2009. doi:10.1109/TIT.2009.2034817.

7 John H. Conway and Richard Guy. *The book of numbers*, pages 258–259. Springer Science & Business Media, 1998. doi:10.1007/978-1-4612-4072-3.

8 Flaviu Cristian. Probabilistic clock synchronization. *Distributed computing*, 3:146–158, 1989. doi:10.1007/BF01784024.

9 M Evans, N Hastings, and B Peacock. Erlang distribution. In *Statistical Distributions, 3rd ed.*, chapter 12, pages 71–73. Wiley, New York, 2000.

10 Daniel R Jeske. On maximum-likelihood estimation of clock offset. *IEEE Transactions on Communications*, 53(1):53–54, 2005. doi:10.1109/TCOMM.2004.840668.

11 Daniel R Jeske and Ashwin Sampath. Estimation of clock offset using bootstrap bias-correction techniques. *Technometrics*, 45(3):256–261, 2003. doi:10.1198/004017003000000078.

12 Leonard Kleinrock. Theory, volume 1, queueing systems, 1975.

13 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. Association for Computing Machinery, New York, 2019. doi:10.1145/3335772.3335934.

14 Mei Leng and Yik-Chung Wu. Low-complexity maximum-likelihood estimator for clock synchronization of wireless sensor nodes under exponential delays. *IEEE Transactions on Signal Processing*, 59(10):4860–4870, 2011. doi:10.1109/TSP.2011.2160857.

15 Alberto Leon-Garcia. *Probability and Random Processes for Electrical Engineering*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1993.

16 Ariel Livshits and Yoram Moses. Probable approximate coordination. *arXiv preprint*, 2023. arXiv:2311.05368.

17 Nancy A Lynch. *Distributed algorithms*, page 466. Elsevier, 1996.

18 Alan Olson and Kang G Shin. Probabilistic clock synchronization in large distributed systems. *IEEE Transactions on Computers*, 43(9):1106–1112, 1994. doi:10.1109/12.312120.

19 Santashil PalChaudhuri, Amit Saha, and David B Johnson. Probabilistic clock synchronization service in sensor networks. *IEEE Transactions on Networking*, 2(2):177–189, 2003.

20 Athanasios Papoulis. *Probability, random variables and stochastic processes*. McGraw-Hill, Columbus, OH, USA, 3rd edition, 1991.

21 Davide Zennaro, Aitzaz Ahmad, Lorenzo Vangelista, Erchin Serpedin, Hazem Nounou, and Mohamed Nounou. Network-wide clock synchronization via message passing with exponentially distributed link delays. *IEEE transactions on communications*, 61(5):2012–2024, 2013. doi:10.1109/TCOMM.2013.021913.120595.

A Probability Space Formulation

We define a probability space $\mathcal{S} = (\Omega, \mathcal{F}, \Pr)$ that supports a countably infinite number of exponential random variables. The sample space $\Omega \triangleq \mathbb{R}_+^{\mathbb{N}}$ is the space of all non-negative sequences. The σ-algebra $\mathcal{F} \triangleq \mathbb{B}(\mathbb{R}_+^{\mathbb{N}}) = \sigma(\Pi_{i=1}^{m}(a_i, b_i) \times \Pi_{m+1}^{\infty}\mathbb{R}_+ : m \in \mathbb{N}, a_i, b_i \in \mathbb{R}_+)$ is the Borel sigma-algebra generated by the Tychonoff topology (see [4]). The probability measure of an event $E = \Pi_{i=1}^{m}(a_i, b_i) \times \Pi_{m+1}^{\infty}\mathbb{R}_+$, is then defined as $\Pr(E) = \Pi_{i=1}^{m} \int_{a_i}^{b_i} \lambda e^{-\lambda t} dt$.

We have chosen this probability space due to the continuous nature of time in our model. In simple terms, $E = \Pi_{i=1}^{m}(a_i, b_i) \times \Pi_{m+1}^{\infty}\mathbb{R}_+$ describes the event that the i-th random variable, for $i \leq m$, takes on a value in the interval (a_i, b_i), and that for all $i > m$, it takes on some arbitrary non-negative value. The number m represents the number of messages sent in runs in which the event E occurs. The event E itself is an m-dimensional box in a countably infinite dimensional space. $\mathcal{F}$ is the σ-algebra generated by closure under complement, and under countable unions and intersections of such events E for any $m \in \mathbb{N}$ and $a_i, b_i \in \mathbb{R}_+$. Additionally, we have chosen the probability measure $\Pr(E) = \Pi_{i=1}^{m} \int_{a_i}^{b_i} \lambda e^{-\lambda t} dt$, i.e., the product of m exponential probability measures, since we assume message arrival times are exponential random variables that are statistically independent.

Let $r \in R(\mathcal{P})$ be a run. We map each run to an element ω in the sample space Ω. We do this by enumerating the messages sent in r as $m_{i,j}^{(1)}, m_{i,j}^{(2)}, \ldots$ and so forth, for all agent-pairs $i, j \in \mathbb{P}$, where message $m_{i,j}^{(k)}$ is the k-th message sent from agent i to agent j. Every message $m_{i,j}^{(k)}$ is associated with an exponential random variable $e_{i,j}^{(k)}$ representing its delay. In the run r, these random variables take on values $\nu_{i,j}^{(k)}$. Thus, the run r is mapped to an $\omega \in \Omega$ that is the sequence of $\nu_{i,j}^{(k)}$ for all $i, j \in \mathbb{P}$ and $k \in \mathbb{N}$. Notice that the number of messages in r must be finite. However, the sequence ω is infinitely long and contains values for message delays that are not sent at all in r. So, for the values of ω that are not associated with a realization of a message delay in r, we choose some arbitrary value. For instance, we may choose the value 1 for all of them, or any other set of non-negative real numbers.

We will mostly use simpler notations whenever the context allows. For example, we define the probability of a run $r \in R(\mathcal{P})$ in which m messages are sent:

▶ **Definition 16.** *Fix a run $r \in R(\mathcal{P})$. Let $e_1, e_2, \ldots, e_m$ be the set of* i.i.d. *exponential random variables associated with the messages sent in r, let $\nu_1, \nu_2, \ldots, \nu_m$ be their realizations in r, and let f_{e_k} be the exponential probability density function (PDF) of e_k. Then, the probability of the run r is $\Pr(r) \triangleq \Pi_{k=1}^{m} f_{e_k}(\nu_k)$.*

Flooding with Absorption: An Efficient Protocol for Heterogeneous Bandits over Complex Networks

Junghyun Lee
Kim Jaechul Graduate School of AI, KAIST, Seoul, Republic of Korea

Laura Schmid
Kim Jaechul Graduate School of AI, KAIST, Seoul, Republic of Korea

Se-Young Yun
Kim Jaechul Graduate School of AI, KAIST, Seoul, Republic of Korea

Abstract

Multi-armed bandits are extensively used to model sequential decision-making, making them ubiquitous in many real-life applications such as online recommender systems and wireless networking. We consider a multi-agent setting where each agent solves their own bandit instance endowed with a different set of arms. Their goal is to minimize their group regret while collaborating via some communication protocol over a given network. Previous literature on this problem only considered arm heterogeneity and networked agents separately. In this work, we introduce a setting that encompasses both features. For this novel setting, we first provide a rigorous regret analysis for a standard flooding protocol combined with the classic UCB policy. Then, to mitigate the issue of high communication costs incurred by flooding in complex networks, we propose a new protocol called Flooding with Absorption (FwA). We provide a theoretical analysis of the resulting regret bound and discuss the advantages of using FwA over flooding. Lastly, we experimentally verify on various scenarios, including dynamic networks, that FwA leads to significantly lower communication costs despite minimal regret performance loss compared to other network protocols.

2012 ACM Subject Classification Theory of computation → Multi-agent learning; Theory of computation → Sequential decision making; Theory of computation → Regret bounds; Networks → Network protocol design

Keywords and phrases multi-armed bandits, multi-agent systems, collaborative learning, network protocol, flooding

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.20

Supplementary Material
Software (Source Code): `https://github.com/nick-jhlee/bandits-network`, archived at `swh:1:dir:41fce125d3e4736eb81fb7771bbf5e795f916c8e`

Funding JH and SY were supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korean government(MSIT) (No.2022-0-00311, Development of Goal-Oriented Reinforcement Learning Techniques for Contact-Rich Robotic Manipulation of Everyday Objects; No.2019-0-00075, Artificial Intelligence Graduate School Program(KAIST)). LS was supported by the Stochastic Analysis and Application Research Center (SAARC) under National Research Foundation of Korea grant NRF-2019R1A5A1028324.

Acknowledgements The authors thank Ulrich Schmid (TU Wien) and the anonymous reviewers for helpful comments and suggestions.

1 Introduction

Exploration-exploitation dilemmas form the basis of many real-life decision-making tasks [51, 41]. In fact, the trade-off between making a choice to either stay with a current action or explore new possibilities appears as a feature in a variety of well-known applications [6, 52].

As a result, the *multi-armed bandit (MAB)* problem, which is designed to reflect dilemmas of this kind, has been intensely studied in a wide range of scenarios [9, 37, 39]. In the baseline setting, an agent must make sequential decisions by choosing from a set of possible actions (the "arms" of the bandit). In the setting of stochastic MABs, each arm gives rewards following an unknown probability distribution [9]. Here, the goal is to minimize the cumulative *regret* over some timespan T, i.e., the difference between the accumulated reward and the reward that arises from choosing only the best arm. To reach this goal, the agent must balance exploring new actions and choosing already tested ones [2, 37, 9].

For applications involving large-scale decentralized decision-making [38], such as online advertising, search/recommender systems, and wireless channel allocation, collaborative multi-agent multi-armed bandits are a natural modeling choice [1, 58, 43, 44, 12, 3, 32, 67, 42]. In this setting, each agent plays their own bandit instance and communicates some information to others to minimize the *group* regret. One common assumption in the literature is that agents share the same set of arms [34, 22, 65]. However, arm homogeneity does not hold in many large-scale systems (e.g., contextual recommender systems), where agents often have heterogeneous arm sets of available actions [68, 15]. For instance, in a distributed recommender system scenario, arms might correspond to the contents shown to users, such as movies, and rewards to user opinions; then, depending on one's location, the set of available movies for each agent may be different due to external constraints such as copyright issues. In this case, it would be desirable for the service provider if all the individual systems with partially overlapping contents collaborate with one another to minimize the group regret.

Another common assumption is that agents are connected by a complete network, where agents can directly communicate with every other agent [10, 68]. However, in real large-scale systems, agents are usually connected via a multi-hop communication network, where only adjacent nodes can exchange messages. To disseminate information to agents at larger distances here, agents need to forward messages. This is typically done by means of a flooding protocol [22, 49, 34, 65] or gossiping protocol [59, 56, 14, 63, 15]; that is, a received message is forwarded to all neighbors or only one randomly selected neighbor, respectively. Such a modeling assumption is ubiquitous in many real-life scenarios involving mobile/vehicular networks or social networks, among others.

Contributions. To the best of our knowledge, the setting of collaborative, *heterogeneous* multi-agent multi-armed bandits communicating over a general *network* has not been investigated yet. Yet, this setting arises in a wide range of real-life applications, e.g. wireless channel allocation, where not all nodes on an underlying network can access the same channels. For such scenarios, one can neither assume fully connected or particularly regular communication topologies, nor homogeneous arm sets. Distinct from previous work on multi-agent bandits, we here aim at efficient *network protocol design* for this novel setting; specifically, we want to design a *simple* alternative to flooding that achieves low communication complexity while retaining minimal loss in regret. Note that our research objective forces us to consider the bandit instances and the communication protocol in an integrated fashion, which is in stark contrast to the approaches used both in the existing bandit and networking literature, and thus contributes to the novelty of our approach.

To address the significant issue of exploding communication complexity of flooding in our setting, we introduce a new lightweight communication protocol for complex networks, called Flooding with Absorption (FwA). Its design principle is inherently coupled with the given bandit instances. We provide theoretical and experimental results showing that this protocol is highly communication-efficient on a wide range of complex networks, yet induces minimal regret performance loss for complex network topologies, even compared

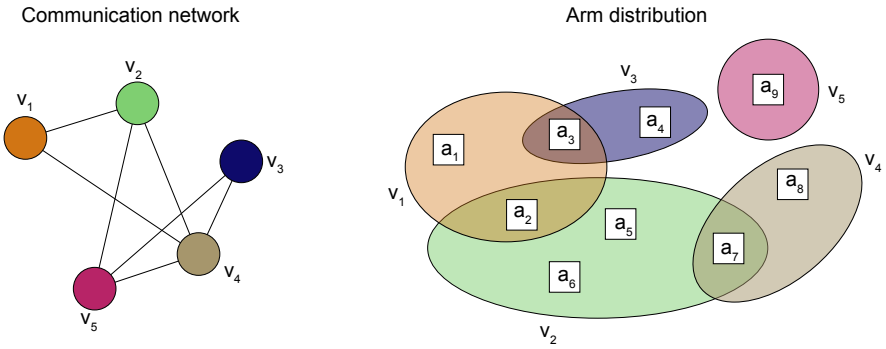


Figure 1 Communication network and arm heterogeneity.

to standard flooding. Using FwA can also help avoid heavy individual link congestion in complex networks. An important practical advantage of our protocol is that it is fully agnostic to the network structure, and can therefore be deployed on dynamically changing networks without any need for fine-tuning.

2 System Model

We now describe our setting of collaborative[1] *heterogeneous* multi-agent multi-armed stochastic bandits over a general communication network; see Figure 1 for an illustration. We assume that there are N agents connected by an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $|\mathcal{V}| = N$. We denote by $\mathcal{N}_{\mathcal{G}}(v)$ be the neighborhood of v in $\mathcal{G}$ *not* including v, and by $\mathcal{G}[S]$ the induced subgraph for $S \subset \mathcal{V}$. Also, for an integer $\gamma \geq 1$, the γ-th order graph power of $\mathcal{G}$, denoted as $\mathcal{G}^{\gamma} = (\mathcal{V}, \mathcal{E}^{\gamma})$, is defined as the graph on $\mathcal{V}$ such that $\{v, w\} \in \mathcal{E}^{\gamma}$ iff $d_{\mathcal{G}}(v, w) \leq \gamma$, where $d_{\mathcal{G}}(v, w)$ is the length of the shortest path in $\mathcal{G}$ connecting v and w. Each agent $v \in \mathcal{V}$ has access to a finite set $\mathcal{K}_v$ of arms with cardinality K_v that they can pull; let $\mathcal{K} = \cup_{v \in V} \mathcal{K}_v$ be the total set of arms of cardinality K.

Following [39], let $\mathcal{M}_{\sigma}$ be a set of σ-sub-Gaussian distributions, and let $\mu : \mathcal{M}_{\sigma} \to \mathbb{R}$ be a function mapping each (reward) distribution to its mean. Each arm $a \in \mathcal{K}$ is associated with an unknown reward distribution $P_a \in \mathcal{M}_{\sigma}$. For simplicity, let $\mu_a := \mu(P_a)$. We note that P_a is independent of the agents' identities, i.e., each agent v, regardless of their arm set $\mathcal{K}_v$, faces the same distribution of rewards for the same arm a (whenever $\mathcal{K}_v$ contains a), and receives an i.i.d. reward from this distribution upon pulling this arm. We denote by $a^v_{\star}$ the best *local* arm for agent v that satisfies $\mu_{a^v_{\star}} > \mu_a$ for all $a \in \mathcal{K}_v \setminus \{a^v_{\star}\}$, and set $\mu^v_{\star} = \mu_{a^v_{\star}}$. The main challenge in the regret analysis is that even for the same arm a, the suboptimality gap may be different across agents containing a.

The execution of all agents proceeds in a sequence of synchronous rounds $t = 1, 2, \ldots$. In each round t, all agents simultaneously (i) pull some arm, (ii) compute and send a message to their neighbors, and (iii) receive and process all messages from their neighbors. From the perspective of agents, let us denote by $\mathcal{V}_a = \{v \in \mathcal{V} : a \in \mathcal{K}_v\} \subseteq \mathcal{V}$ the set of agents having action a, and let $\mathcal{V}_{-a} \subseteq \mathcal{V}_a$ be the set of agents containing a as a *suboptimal* arm, i.e., $a \neq a^v_{\star}$.

[1] We remark that we do *not* consider any collisions [46, 40, 65] where two neighbors pulling the same arm do not affect their observed rewards in any way. Rather, we focus on the collaborative setting where the agents are encouraged to cooperate with one another by sharing their own observations.

As done in the classic work on regret minimization in collaborative multi-agent bandits [34, 68, 49], our goal is to minimize the expected *group* regret at time horizon T, $\mathbb{E}[R(T)]$, defined as:

$$\mathbb{E}[R(T)] := \sum_{v \in \mathcal{V}} \mathbb{E}[R^v(T)], \quad \mathbb{E}[R^v(T)] := \sum_{a \in \mathcal{K}_v} \Delta_a^v \mathbb{E}[N_a^v(T)], \tag{1}$$

where $\Delta_a^v := \mu_\star^v - \mu_a$ is the agent-specific gap of arm a and $N_a^v(t)$ is the number of times agent v plays the arm a up to time t.

We note that due to the two sources of heterogeneity in our setting, the benefit one receives from collaboration differs; furthermore, the same arm a may be optimal for some agents but suboptimal for others. Of course, even so, we expect that agents communicating and collaborating should lead to a speed-up (in terms of the group regret) compared to the baseline without any information sharing. The question is then how much speed-up one could get from collaboration under our heterogeneous system model. An additional prominent issue for realistic applications and, in particular, *complex* networks concerns the communication complexity of the information-sharing protocol used. Designing communication protocols with *low* communication complexity, defined as the number of messages sent and forwarded, is of paramount importance to not overshadow the improvement in regret due to collaboration.

3 Flooding

As is common in much of the previous literature, we will focus on agents that individually run the classic upper-confidence bound (UCB) policy [2, 34, 68] under which each agent pulls the arm associated with the maximum of the so-called UCB index, which is the sum of empirical reward (up to time t) and an additional bonus term. Agents can, and should, take advantage of the distributed setting by communicating pulled arms and received rewards amongst each other over the underlying communication network. This is not straightforward to implement or analyze, given that the agents' arm sets are all different, and thus, even when broadcasting over multiple hops by flooding the network is available, the speed-up gained from this is not immediate. In this section, we review (and reformulate) the standard flooding protocol for our setting and discuss its properties.

3.1 Flooding

FLOODING (broadcasting) lets each agent send all messages to all its neighbors in every round, with the number of times the message is forwarded limited by the *time-to-live* (TTL) γ [13, 60, 64, 55]. To account for potential loops in the network and avoid a broadcast storm [61, 66], we explicitly use a sequence number-controlled flooding (SNCF) variant. The pseudocode of FLOODING along with each agent's UCB algorithms is presented in Algorithm 1 with *absorb* $= False$.

FLOODING proceeds as follows: In each round t, each agent v pulls an arm $a^v(t)$ that has the highest upper confidence bound (line 8). Note that $M_a^v(t)$ is the number of pulls of arm a available to agent v by time t, and $\hat{\mu}_a^v$ is the estimate of μ_a made by agent v at time t. In both estimates, agent v uses all observations available to v by time t, including the messages relayed to them. Having received the corresponding reward $X^v_{a^v(t)}(t)$ from pulling arm $a^v(t)$, agent v creates a message

$$m = \langle a^v(t), X^v_{a^v}(t), \textsc{Hash}(v, a^v(t), X^v_{a^v}(t)), v, \gamma \rangle, \tag{2}$$

and pushes it to $\mathcal{M}^v$, its current queue of messages to be sent (line 11). After UCB has been completed, each agent sends and receives messages to and from its neighbors (lines 14–21).

Algorithm 1 FLOODING, FWA WITH UCB.

Input: $\gamma \in \mathbb{N}$, $\alpha > 0$, *absorb* $\in \{True, False\}$, $f : \mathbb{N} \to [1, \infty)$

1 Initialize $\mathcal{M}^v = queue()$, $\mathcal{H}^v = queue(maxlen = \gamma N)$;
2 $M_a^v(0) = 0, \hat{\mu}_a^v = 0$ for all $(v, a) \in \mathcal{V} \times \mathcal{K}$;
3 **for** $t \in [T]$ **do**
4 **for** $v \in \mathcal{V}$ **do**
5 **if** $t \leq K_v$ **then**
6 play the t-th arm of $\mathcal{K}_v$;
7 **else**
8 play the arm $a^v(t)$ chosen as follows:
$$a^v(t) = \arg\max_{a \in \mathcal{K}_v} \left\{ \hat{\mu}_a^v(t) + \sqrt{\frac{2 \log f(t)}{M_a^v(t)}} \right\};$$
9 **end**
10 $hash \leftarrow \text{HASH}(v, a^v(t), X^v(t))$;
11 $\mathcal{M}^v.push(\langle a^v(t), X^v(t), hash, v, \gamma \rangle)$;
12 **end**
13 **for** $v \in \mathcal{V}$ **do**
14 **while** $\widetilde{\mathcal{M}}^v$ *is not empty* **do**
15 $m \leftarrow \widetilde{\mathcal{M}}^v.pop()$;
16 **for** $w \in \mathcal{N}_{\mathcal{G}}(v) \setminus \{m(3)\}$ **do**
17 $m' \leftarrow copy(m)$;
18 **if** $m'(2) \notin \mathcal{H}^w$ **then**
19 $\mathcal{H}^w.push(m'(2))$;
20 `Receive` $(w, m',$ *absorb*$)$;
21 **end**
22 delete m'
23 **end**
24 delete m;
25 **end**
26 **end**
27 **end**

Algorithm 2 Subroutines.

1 **Function** `Receive`$(w,$ $m,$ *absorb*$)$**:**
2 **if** $m(0) \in \mathcal{K}_w$ **then**
3 $M_{m(0)}^w(t) \leftarrow M_{m(0)}^w(t-1) + 1$,
$\hat{\mu}_{m(0)}^w(t) \leftarrow \frac{M_{m(0)}^w(t-1)\hat{\mu}_{m(0)}^w(t-1) + m(1)}{M_{m(0)}^w(t)}$;
4 **if** *absorb is True* **then**
5 delete m;
6 **else**
7 `Store` (w, m);
8 **end**
9 **else**
10 `Store` (w, m);
11 **end**
12 **Function** `Store`(w, m)**:**
13 $m(4) \leftarrow m(4) - 1$;
14 **if** $m(4) < 0$ **then**
15 delete m;
16 **else**
17 $m(3) \leftarrow w$;
18 $\mathcal{M}^w.push(m)$;
19 **end**

Our message $m = \langle m(0), m(1), m(2), m(3), m(4) \rangle$ consists of the following components: $m(0)$ and $m(1)$ are the arm pulled by agent v and the reward received at time t, respectively. $m(2)$ is a hash value of the originating agent v, the arm pulled, and the obtained reward that acts as a unique identifier of the message. Our protocol uses $m(2)$ to control flooding by avoiding routing loops that can lead to broadcast storms and improper bias in the reward estimations (the estimation protocol is shown in line 3 of Algorithm 2). Each agent v keeps track of the hash values of messages that they have seen until time t via a *queue*[2] of size γN, denoted as $\mathcal{H}^v$. If an already-seen message comes in (line 18), that message is deleted on arrival (line 22). The memory length of γN is the worst-case space complexity that arises from keeping track of all messages from all agents for the last γ time steps, as all messages can be forwarded at most γ times. $m(3)$ is the agent that last forwarded the message; if the receiver w passes on m, they replace $m(3)$ with w and forward m to all neighbors except the originator $m(3)$ (line 16). This prevents messages from echoing after one hop. $m(4)$ keeps track of the remaining life span of the message (TTL), which is initialized to γ. It is decayed by 1 every time a message is forwarded, and the message is discarded when TTL reaches 0. We note that $\gamma = 1$ is equivalent to INSTANT REWARD SHARING (IRS) [34], where each agent only sends its message to its neighbors, and any message containing arm a that is sent to agents not containing a becomes void.

We assume that the nodes have no knowledge of the network topology, or who their neighbors are, which is a realistic assumption in complex networks, and wireless networks. This is in contrast to some of the previous works, e.g., [49], which assumes that each agent

[2] The queue operations used in the algorithm ($pop()$, $push()$, $flush()$) are defined as usual [21].

knows its neighborhood in $\mathcal{G}^\gamma$, which essentially bypasses any issues regarding communication complexity. Specifically, this also abstracts away any difficulties arising from delayed messages traveling along different paths.

3.2 Group Regret Analyses of Flooding

The regret bound of stochastic bandits generally depends on a problem-dependent quantity [2] that quantifies the difficulty of learning. For instance, for a single-agent multi-armed stochastic bandit, the regret bound scales as $\sum_{a\in\mathcal{K}\setminus\{a^\star\}} \frac{\log T}{\Delta_a}$, where Δ_a is the gap between the mean rewards of best arm $a^\star$ and the suboptimal arm a. The intuition is that if the mean rewards are similar, one would need a much tighter confidence interval to identify the optimal arm, forcing one to pull a many more times, precisely inversely proportional to its reward gap. This is known to be asymptotically optimal [37].

In our setting, we would expect our problem-dependent quantity to depend on both the underlying network topology and arm distribution. To see this, given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we first recall some graph-theoretic quantities [7]:

▶ **Definition 3.1.** *The* **clique covering number** *of $\mathcal{G}$, denoted as $\theta(\mathcal{G})$, is the smallest size of a partition of $\mathcal{V}$ such that each part induces a clique. Any such partition (not necessarily minimum) is called a* **clique cover**. *The* **independence number** *of $\mathcal{G}$, denoted as $\alpha(\mathcal{G})$, is the maximum size of a subset of $\mathcal{V}$ that induces no edges.*

We now define our problem-dependent quantity Δ_a^γ as follows:

$$\Delta_a^\gamma := \max_{\mathcal{C}_{-a}} \left(\sum_{C\in\mathcal{C}_{-a}} \left\{ \frac{2}{\min_{v\in C} \Delta_a^v} - \frac{1}{\max_{v\in C} \Delta_a^v} \right\} \right)^{-1}, \tag{3}$$

where $\Delta_a^v = \mu_\star^v - \mu_a$ is the agent-specific suboptimality gap of arm a, and $\max_{\mathcal{C}_{-a}}$ is taken over all possible clique covers $\mathcal{C}_{-a}$ of $[\mathcal{G}^\gamma]_{-a} := \mathcal{G}^\gamma[\mathcal{V}_{-a}]$. If $\mathcal{V}_{-a} = \emptyset$, we set $\Delta_a^\gamma = 0$.

We now present the nonasymptotic regret upper bound for FLOODING:

▶ **Theorem 3.2.** *Algorithm 1 with absorb = False, $f(t) = t^\alpha$, $\alpha > \max\left(\frac{1}{2}, \frac{2\sigma^2}{\gamma+1}\right)$, and $\gamma \in \{1, \cdots, \mathrm{diam}(\mathcal{G})\}$ achieves the group regret upper bound*

$$\mathbb{E}\left[R(T)\right] \le \sum_{\substack{a\in\mathcal{K}\\ \Delta_a^\gamma>0}} \frac{4\alpha\log T}{\Delta_a^\gamma} + b(\gamma) + \sum_{a\in\mathcal{K}} f_a(\gamma), \tag{4}$$

where

$$b(\gamma) := \left(\frac{\alpha+1/2}{\alpha-1/2}\right)^2 \frac{8(\gamma+1)}{\log\frac{(\gamma+1)(\alpha+1/2)}{4\sigma^2}} \sum_{a\in\mathcal{K}} \sum_{v\in\mathcal{V}_a} \Delta_a^v, \quad f_a(\gamma) = \sum_{v\in\mathcal{V}_{-a}} \Delta_a^v \min\left(2\gamma, \frac{4\alpha\log T}{(\Delta_a^v)^2}\right).$$

The complete proof, deferred to Appendix B, uses a clique covering argument and an Abel transformation. See Appendix C.1 for a discussion of the main technical challenges when proving the regret bound for our setting, compared to previously considered settings.

By choosing $\mathcal{C}_{-a}$ as the minimum clique cover for each a in the definition of Δ_a^γ, a simplified, asymptotic regret bound can be deduced:

▶ **Corollary 3.3.** *When* $\max\left(b_{\alpha,\gamma,\sigma}, \sum_{a\in\mathcal{K}} f_a(\gamma)\right) = o(\log T)$,

$$\limsup_{T\to\infty} \frac{\mathbb{E}\left[R(T)\right]}{\log T} \le \sum_{\substack{a\in\mathcal{K}\\ \Delta_a^\gamma>0}} \frac{4\alpha}{\Delta_a^\gamma} \le \sum_{\substack{a\in\mathcal{K}\\ \tilde{\Delta}_a>0}} \frac{8\alpha\theta([\mathcal{G}^\gamma]_{-a})}{\tilde{\Delta}_a}, \tag{5}$$

where $\tilde{\Delta}_a := \min_{v\in\mathcal{V}_{-a}} \Delta_a^v$.

Note that $\tilde{\Delta}_a$ is the suboptimality gap introduced in Yang et al. [68], where they studied the setting of heterogeneous bandits on a fully-connected network. When $\mathcal{G}$ is a fully connected network, $\theta(\cdot) = 1$, and we recover their regret bound, with an improved constant in α.

Since our general setting also applies to the restricted cases considered in the previous literature, we can compare our regret bounds to existing ones. In the same setting without collaboration, the group regret scales as $\sum_{a \in \mathcal{K}} \frac{|\mathcal{V}_{-a}| \log T}{\tilde{\Delta}_a}$; thus when $\tilde{\Delta}_a$ is considered to be constant, the regret always scales linearly in N. Compared to this, depending on the network, the regret bound of FLOODING scales with the clique covering number of subgraphs of $\mathcal{G}^\gamma$, which is usually strictly less than N, and in some cases, even sublinear.

Similarly, our regret bounds and our problem-dependent difficulty quantity Δ_a^γ also generalize previous literature on collaborative multi-agent multi-armed bandits. When the network is a clique, we recover the regret bound presented in [68] with matching $\log T$ dependency and an improved leading coefficient[3]. In the homogeneous agents setting with a general network topology [34, 49], Δ_a^γ reduces to $\frac{\Delta_a^{Kolla}}{\theta(\mathcal{G}^\gamma)}$, where Δ_a^{Kolla} is the suboptimality gap as defined in [34], satisfying $\Delta_a^{Kolla} := \Delta_a^v$ for all $v \in \mathcal{V}$. As $\frac{1}{\theta(\mathcal{G}^\gamma)}$ is independent of the arm a, we have shown that our Δ_a^γ successfully generalizes the suboptimality gap of [34]. When $\gamma = \mathrm{diam}(\mathcal{G})$, we have that $\Delta_a^\gamma = \Delta_a^{Kolla}$ as $\theta(\mathcal{G}^\gamma) = 1$, which results in the same regret bound as in [34].

When $\gamma = 1$ (IRS), it can be observed that IRS and FWA coincide, yet our bound is a bit worse compared to [34], whose bound depends on $\alpha(\mathcal{G})$. We believe that such a gap is an inherent artifact of our proof, and we leave closing this gap to future work. Also, we should remark that the difference between the two regret bounds for IRS is generally small, as the gap depends on the covering gap $\theta(\mathcal{G}) - \alpha(\mathcal{G})$, which is known to be small for many classes of graphs, and zero for perfect graphs [27]; see [28, 53] for some recent advances.

3.3 Drawbacks of Flooding

Flooding leads to optimal information dissemination, and thus significantly improves the group regret. Yet, it is very expensive in terms of communication complexity, defined as the *cumulative* number of messages sent by all agents [68]. Indeed, for $\gamma = \mathrm{diam}(\mathcal{G})$, the worst-case communication complexity is $\gamma N|\mathcal{E}|T$, which is attained when every message created by every agent up to time T is being passed around at every edge. This can quickly congest the network. One naïve way of controlling the communication complexity is to set the TTL, γ, to a low enough value. However, in our setting, the trade-off between communication complexity and group regret is not trivial due to the arm heterogeneity; for instance, IRS [34, 49], i.e., $\gamma = 1$, has a lower message complexity $N|\mathcal{E}|T$ but often does not result in good regret guarantees, as immediate neighbors may not share any arms. On the other hand, (uniform) gossiping algorithms for bandit problems [14, 56, 57] suffer from large latencies on networks with sparse links [29, 11].

It is thus desirable to have a simple communication protocol with good regret guarantees when combined with the UCB policy (compared to FLOODING) and low communication complexity. With this, we can target more complex network structures commonly found in real-life applications. For such settings, we introduce a new protocol that interpolates between the communication-efficient nature of IRS and the regret of FLOODING by using the intrinsic heterogeneity of the system caused by network topology and arm distributions.

[3] Theorem 2 of [68] requires $\alpha > 2$, and thus with proper scaling, it can be seen that our coefficient is 8α while their coefficient is 24α.

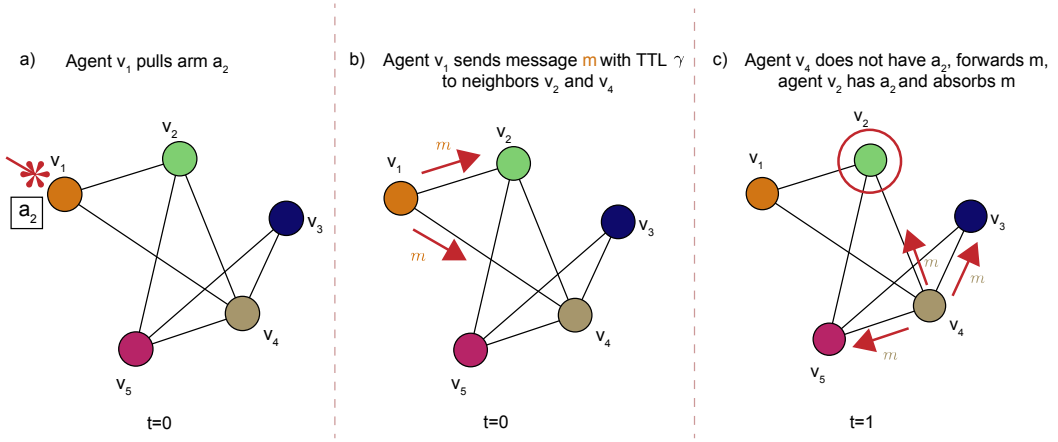


Figure 2 Flooding with Absorption (FwA). **a,** An agent (v_1) pulls one of its arms (a_2). **b,** v_1 sends a message m to its neighbors, with a TTL γ. **c,** Since one receiver of the message (v_4) does not have a_2 in its arm set, they forward m to their neighbors except the originator v_1. The other receiver (v_2) has arm a_2 in their arm set, and thus it absorbs m.

4 A New Efficient & Effective Protocol on Complex Networks: Flooding with Absorption

In this section, we propose a new approach, which we call **Flooding with Absorption (FwA)** (Figure 2), whose pseudocode is shown in Algorithm 1 with *absorb* = *True*. In contrast to FLOODING, once a message (some copy of it to be precise) containing arm a reaches an agent whose arm set includes the arm a, the agent *absorbs* that message, i.e., does not forward it any further. Additionally, as in FLOODING, we retain the TTL γ, meaning that if a message originating at time t has not found an absorbing agent until $t' = t + \gamma$, it gets discarded. It is also dropped if the message hits a "dead end," i.e., a leaf node.

This seemingly small difference to FLOODING is critical in ensuring low communication complexity, as it prevents messages from circulating for too long. We note that FWA is somewhat reminiscent of the well-studied replication-based epidemic- and other controlled flooding algorithms [62, 55, 45, 24], which were designed for various networking applications, e.g., ad-hoc networks. Our FWA protocol distinguishes itself by *using* the inherent heterogeneity of agents *without* any explicit tuning or need for solving NP-hard combinatorial problems [45]. Furthermore, the goal of FWA is to disseminate information, which is generated at each timestep, to nodes that may benefit from it for its learning, *not* to route packets from an arbitrary source to an arbitrary destination. Despite some outward protocol similarity, it hence also differs from classic P2P systems such as Gnutella [47, 48], where there is no such intrinsic correlation between sender and receiver. In the bandit setting, FwA is advantageous because the sender and final receiver share the arm in question.

▶ Remark 4.1. Each agent must have a sufficiently large memory buffer to store the messages to be sent in the next round and previously seen message identifiers. As all messages expire after γ rounds, this memory requirement is at most γN. Also, we note that the communication complexity of FWA ranges from $N|\mathcal{E}|T$ to $\gamma N|\mathcal{E}|T$, depending on the underlying network topology and the arm distribution.

4.1 Group Regret Bound of FwA

As FwA is algorithmically similar to Flooding, their regret bounds are also somewhat similar. To formalize this, we first consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and let $\boldsymbol{c} : \mathcal{V} \to 2^{\mathcal{K}}$ be a multi-coloring with overlap allowed, i.e., it may be that $\boldsymbol{c}(v) \cap \boldsymbol{c}(w) \neq \emptyset$ for $\{v, w\} \in \mathcal{E}$. Let $a \in \mathcal{K}$ and $v, w \in \mathcal{V}$ be such that $a \in \boldsymbol{c}(v) \cap \boldsymbol{c}(w)$.

▶ **Definition 4.2.** *A path $v_0 v_1 \cdots v_m$ (of length m) is said to* **a-free** *if $a \notin \bigcup_{i \in [m-1]} \boldsymbol{c}(v_i)$, where $[m] = \{0, 1, ...m\}$.*

▶ **Definition 4.3.** *For $\gamma \geq 1$ and $a \in \mathcal{K}$, we define the* **$(a, \boldsymbol{c})$-non-blocking graph power of γ-th order of $\mathcal{G}$,** *denoted as $\mathcal{G}^{\gamma}_{(a,\boldsymbol{c})}$, as a graph on $\mathcal{V}$ with the edge set $\mathcal{E}^{\gamma}_{(a,\boldsymbol{c})}$ such that $\{v, w\} \in \mathcal{E}^{\gamma}_{(a,\boldsymbol{c})}$ iff there exists an a-free path from v to w in $\mathcal{G}$ of length at most γ.*

▶ Remark 4.4. Def. 4.3 is similar to *color-avoiding percolation (CAP)* in statistical physics [35, 36, 33], albeit there are several differences. We consider a multi-coloring $\boldsymbol{c} : v \mapsto \mathcal{K}_v$, while CAP is only studied for a single color per vertex. Also, CAP only considered the criticality of the connectivity, while in our case, the performance of our algorithm depends on specific graph invariants (e.g., chromatic number) of color-dependent subgraphs.

Defining the suboptimality gap Δ_a^{FwA} for FwA as

$$\Delta_a^{FwA,\gamma} := \max_{\mathcal{C}^{\gamma}_{(a,\boldsymbol{c})}} \left(\sum_{C \in \mathcal{C}^{\gamma}_{(a,\boldsymbol{c})}} \left\{ \frac{2}{\min_{v \in C} \Delta_a^v} - \frac{1}{\max_{v \in C} \Delta_a^v} \right\} \right)^{-1}, \tag{6}$$

where $\min_{\mathcal{C}^{\gamma}_{(a,\boldsymbol{c})}}$ is over all possible clique covers $\mathcal{C}^{\gamma}_{(a,\boldsymbol{c})}$ of $\mathcal{G}^{\gamma}_{(a,\boldsymbol{c})}$, it is easy to see that the following theorem holds:

▶ **Theorem 4.5.** *With absorb $= True$, Theorem 3.2 holds with Δ_a^{γ} replaced by $\Delta_a^{FwA,\gamma}$.*

Similarly, with an appropriate choice of the clique cover, we have the following simplified asymptotic regret bound:

▶ **Corollary 4.6.** *With the same assumption as in Theorem 3.2, we have*

$$\limsup_{T \to \infty} \frac{\mathbb{E}[R(T)]}{\log T} \leq \sum_{\substack{a \in \mathcal{K} \\ \Delta_a^{\gamma} > 0}} \frac{4\alpha}{\Delta_a^{FwA,\gamma}} \leq \sum_{\substack{a \in \mathcal{K} \\ \tilde{\Delta}_a > 0}} \frac{8\alpha\theta\left([\mathcal{G}^{\gamma}_{(a,\boldsymbol{c})}]_{-a}\right)}{\tilde{\Delta}_a}, \tag{7}$$

where $\tilde{\Delta}_a = \min_{v \in \mathcal{V}_{-a}} \Delta_a^v$.

As $\mathcal{G}^{\gamma}_{(a,\boldsymbol{c})}$ is always a subgraph of $\mathcal{G}^{\gamma}$, it can be easily seen that the regret upper-bound of Flooding is always better than that of FwA. But as we will demonstrate later, at the price of *slightly* worse regret, FwA obtains significantly better communication complexity than Flooding. Corollary 3.3 and 4.6 imply that the gap in the asymptotic regret upper-bounds of Flooding and FwA roughly scales with $\sum_{\substack{a \in \mathcal{K} \\ \tilde{\Delta}_a > 0}} \frac{\delta_a}{\tilde{\Delta}_a}$, where $\delta_a := \theta\left([\mathcal{G}^{\gamma}_{(a,\boldsymbol{c})}]_{-a}\right) - \theta\left([\mathcal{G}^{\gamma}]_{-a}\right)$.

To get some intuition, we consider two extreme cases. First, suppose that the arms are so heterogeneous that no agents of distance at most γ share arm a. In this case, we have $\mathcal{G}^{\gamma}_{(a,\boldsymbol{c})} = \mathcal{G}^{\gamma}$, and $\delta_a = 0$. Now suppose that all agents have the same arm set, i.e., $\mathcal{K}_v = \mathcal{K}$ for all $v \in \mathcal{V}$, in which case FwA is equivalent to IRS, i.e., messages do not get forwarded beyond direct neighbors. Hence, we have that $\mathcal{G}^{\gamma}_{(a,\boldsymbol{c})} = \mathcal{G}$ for all $\gamma \geq 1$ and $a \in \mathcal{K}$, i.e., $\delta_a = \theta([\mathcal{G}]_{-a}) - \theta([\mathcal{G}^{\gamma}]_{-a})$. Thus, for small γ's and $\mathcal{G}$ with *large* average path length [5] between agents containing a, δ_a is small.

4.2 Advantages of Flooding with Absorption

We now informally argue the advantages of using Flooding with Absorption over other protocols such as Flooding or IRS when we run it on complex network topologies.

Interpolation between IRS and Flooding. FwA naturally interpolates between IRS and Flooding in terms of information propagation, which is advantageous on complex networks that are not particularly regular, e.g., those with both dense and sparse regions. In dense parts of the network, where many nodes share arms, FwA is closer to IRS: a message containing the shared arm and its reward gets absorbed quickly. On the other hand, in regions of the network where the arm that a particular node pulled is rare, FwA acts like Flooding with $\gamma \gg 1$, thereby ensuring that agents at a larger distance get information that is relevant. We additionally note that setting the TTL to larger values in FWA will always be less costly than doing so in Flooding, as the probability of congestion is much smaller.

Comparable Regret Guarantees. As FwA acts as a mix of IRS and Flooding, its regret should be bounded by IRS (where messages get absorbed in just one step) from above, and Flooding (where messages are not absorbed until the TTL expires) from below. The combination of Theorems 3.2 and 4.5 gives us an expression for the gap between the regret upper bounds of Flooding and FwA. From this, we can conclude that for the regret gap between FwA and Flooding to be small, either the graph is so sparse that the average path length [5] between agents *containing a* is large, or the graph is dense but the arm distribution is sparse enough such that the same property holds. We emphasize that although the gap may be nonzero, the exploding communication complexity of Flooding demonstrates a clear trade-off between performance and communication complexity. On the other hand, it is also expected that FwA will outperform (uniform) gossiping in terms of the regret. In fact, in networks with sparse links connecting very dense network regions, the probability that a gossiping protocol hits the sparse link before the TTL expires can be arbitrarily small.

Communication Efficiency. Having messages absorbed by agents that can profit from its information implies that the FwA protocol completely falls back to the baseline Flooding algorithm only in the case of a network where particular arms are very rare. This means that if there is a ball of radius γ in the network in which two agents share an arm, the communication complexity of FwA will already be lower than that of Flooding, $O(N \cdot |\mathcal{E}| \cdot \gamma \cdot T)$. In networks of high density and few arms, the communication complexity of FwA will be close to that of IRS, i.e., lowered by a factor γ. Moreover, due to the arm-dependent absorptions, we expect that FwA will result in much lesser number of messages sent across each individual link per round. Hence, FwA has the advantage of being able to mitigate network overload and heavy link congestion without much overhead or the need to resort to gossiping protocols, which is particularly salient for applications such as large-scale and wireless networks.

No tuning requirements. FwA also has an important practical advantage: it has no tunable parameters beyond the TTL γ, which means it is close to network agnostic. This is in contrast to protocols like probabilistic flooding [54], which stops message propagation with some constant probability q. While such protocols can also reduce communication complexity to a scalable degree, the "optimal" stopping probability q is highly instance-dependent, making them quite hard to use in unknown, real-life networks, in particular, dynamically changing networks. FwA, on the other hand, can effectively deal with such instances, which we verify in Section 6.

5 Regret Lower Bound

We consider a decentralized[4] policy $\Pi = (\pi^v)_{v\in\mathcal{V}}$, where $\pi^v : [T] \to \mathcal{P}(\mathcal{K})$ is the agent-wise policy followed by agent v, possibly affected by other policies and the history. For the regret lower bound, we consider a rather general class of policies satisfying the following property, which has been widely adapted in bandit literature [37, 34, 22]:

▶ **Definition 5.1.** Π *is said to be* **individually consistent** *if, for any agent v and any $a \in \mathcal{K}_{-v}$, we have that $\mathbb{E}[N_a(T)] = o(T^c)$, $\forall c > 0$, where $N_a(T) := \sum_{v\in\mathcal{V}_a} N_a^v(T)$.*

One obtains the following regret lower bound:

▶ **Theorem 5.2.** *For any individually consistent policy* Π*,*

$$\liminf_{T\to\infty} \frac{\mathbb{E}[R(T)]}{\log T} \geq \sum_{\substack{a\in\mathcal{K}\\ \tilde{\Delta}_a>0}} \frac{\tilde{\Delta}_a}{\inf_{P\in\mathcal{M}_\sigma}\left\{D_{\mathrm{KL}}(P_a, P) : \mu(P) - \mu(P_a) > \tilde{\Delta}_a\right\}}, \tag{8}$$

where $\mathcal{M}_\sigma$ is the set of σ-sub-Gaussian distributions with mean μ, and $\tilde{\Delta}_a = \min_{v\in\mathcal{V}_{-a}} \Delta_a^v$.

When $\mathcal{M}_\sigma = \left\{\mathcal{N}(\mu, \sigma^2) : \mu \in \mathbb{R}\right\}$, we obtain:

$$\liminf_{T\to\infty} \frac{\mathbb{E}[R(T)]}{\log T} \geq \sum_{\substack{a\in\mathcal{K}\\ \tilde{\Delta}_a>0}} \frac{2\sigma^2}{\tilde{\Delta}_a}. \tag{9}$$

The proof is immediate from the classic change-of-measure argument for cooperative multi-agent bandit setting [22, 68]. Note that this asymptotic lower bound matches our asymptotic regret upper bound for both Flooding and FWA (Theorem 4.5, Cor. 4.6) up to some graph topology and arm distribution-dependent constants; see Appendix C.2 for a more detailed discussion.

6 Experimental Results

In this section, we experimentally compare FWA to several existing algorithmic solutions. The experiments were conducted on three random graph models with $N = 100$ nodes: the Erdős-Rényi model (ER) [23, 26], the Barabási-Albert model (BA) [4], and the stochastic block model (SBM) [31]. We use the following hyperparameters when generating the random graphs: for ER, the edge probability is set to $p_{ER} = 0.03$; for BA, the preferential attachment constant is set to 2; for SBM, we consider 4 clusters, each with $N/4$ nodes, with intracluster and intercluster edge probabilities set to 0.3 and 0.003, respectively.

We set the total number of arms to $K = 50$, and the number of arms per agent to be $k = 20$. We sample sets of size k as arm sets for all the agents, uniformly at random. The arm rewards follow Gaussian distributions, with the corresponding means uniformly sampled from $[0, 1]$ and fixed variance $\sigma^2 = 1$. For all experiments, the overall arm distribution among the agents and the reward distributions are fixed.

We compare the baseline UCB with no cooperation between agents (baseline), Flooding, Probabilistic Flooding (Prob. Flooding) [54], (uniform) Gossip, IRS, and our FWA. For the Gossip algorithm, we assume that each agent forwards messages to only one random

[4] see Appendix A of [22] for the precise measure-theoretic definition of decentralized policies.

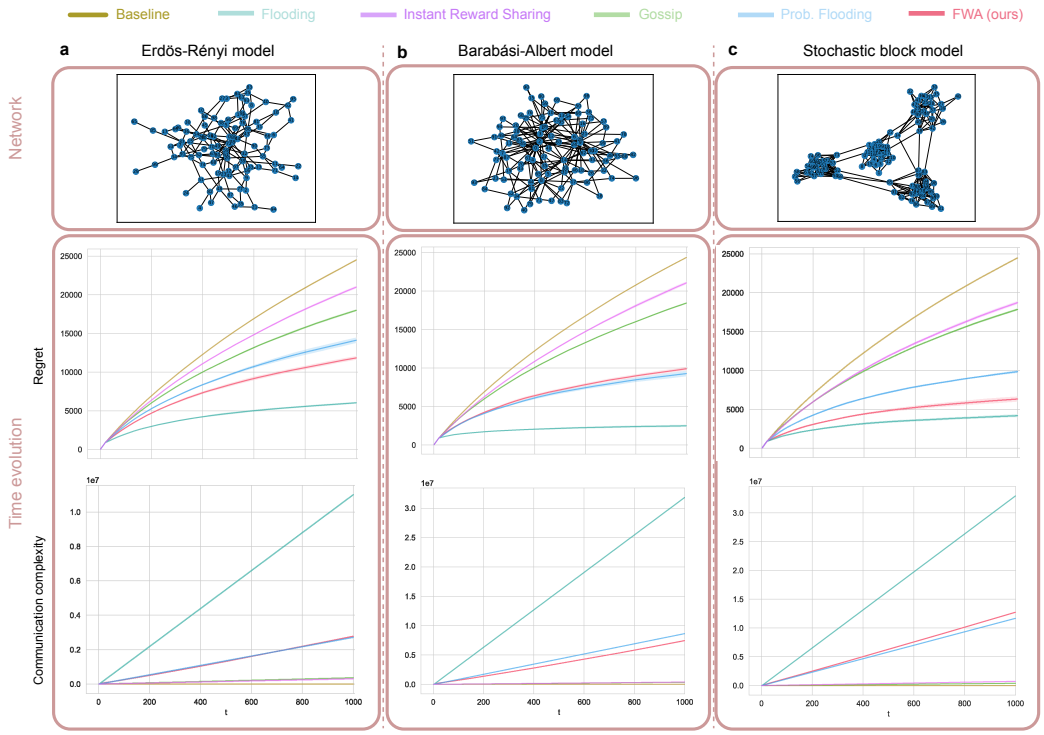


Figure 3 Comparing group regret and (cumulative) communication complexity across different topologies and protocols. Note that FwA gives a good trade-off between regret and communication complexity.

neighbor at a time. For Prob. Flooding, assuming that the learner has no prior knowledge of the communication network, we fix $q = 0.5$ to provide a fairer comparison. We again emphasize that FwA does *not* require any tuning of hyperparameters other than the TTL.

All experiments are repeated 10 times, with time horizon $T = 1000$. We set TTL to be $\gamma = 4$, as our network instances' diameters are $8, 5, 7$ for ER, BA, and SBM, respectively, and we want our γ to be strictly lower than all three values for meaningful results. All codes were written in Python, and we made heavy use of the NetworkX package [30].

6.1 Baseline comparison: Group Regret and Communication Complexity

We first compare the group regret and the communication complexity over time, i.e., the (cumulative) total number of messages sent, for all considered protocols. The results are shown in Figure 3. As expected, Flooding achieves the best regret out of the tested protocols, but its communication complexity is the worst. Despite this, the important observation is that our FwA achieves second-best regret, arguably close to that achieved by Flooding, with a significantly reduced communication complexity. Moreover, when compared to Prob. Flooding, FwA exhibits a better tradeoff between regret and communication: with a similar communication complexity, FwA achieves a better regret, or at least at par (for the BA model). We also experimented with Prob. Flooding of other stopping probabilities (not shown here) and observed that they tend to show worse trade-offs between regret and communication complexity. This shows that our proposed FwA protocol is a viable alternative to Flooding if one needs reduced communication complexity and good regret, uniformly across various network topologies.

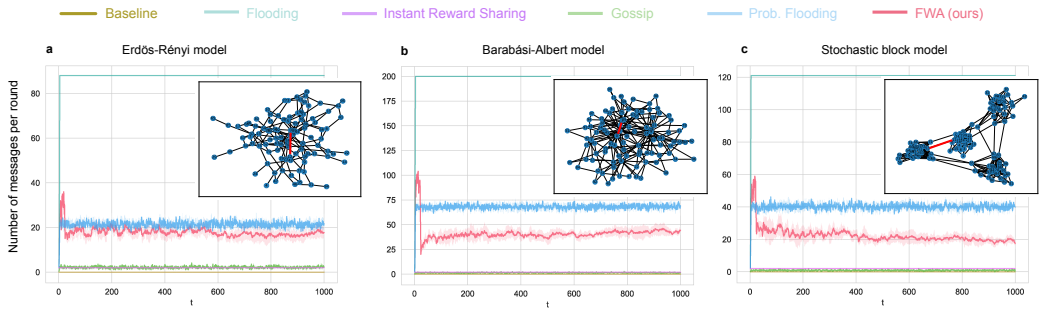


Figure 4 FWA significantly decreases congestion on sparse network links. We find that, in comparison with other protocols, FWA results in a reduced number of messages sent over such a sparse link (highlighted in the networks).

6.2 Link congestion

In a setting where new messages are constantly produced by every agent (as each pulls an arm at each time step), one of the potential issues is link congestion caused by a large number of messages passing through bottleneck links. This can lead to significantly decreased performance and undesirable latency effects - messages may be queued with limited memory in reliable link protocols, or automatically discarded in non-reliable link protocols once more messages are being sent than the link can handle. In Figure 4, we visualize the number of messages sent over a particular link, again with TTL $\gamma = 4$. We chose a random link for ER and BA. For SBM, we chose a "sparse" link that connects two dense clusters. Out of all the considered protocols, FWA results in the largest reduction of messages per round while providing good regret; specifically, for ER, BA, and SBM, FwA provides about $77\%, 80\%, 83\%$ reduction compared to Flooding, respectively. We note that the reduction is larger even compared to Prob. Flooding - even though Prob. Flooding can occasionally have slightly better overall communication complexity (at worse regret). This implies that our protocol exhibits significant benefits regarding individual network link congestion, which would help us avoid latency effects in real-life network applications.

One interesting observation is that FWA produces a spike in the number of messages in the early phase for all network topologies. This is due to the design of the UCB algorithm; in the early phase, most of the agents are exploring the arms, and thus the messages are somewhat "diverse". But as soon as the agents identify potentially best arms, the arm indices of the messages start to stabilize. They become less diverse, implying that from then on, absorption occurs more frequently under FwA.

6.3 Dynamic networks

The advantage of our protocol is especially pronounced when we consider *dynamically changing networks*, specifically, edge-Markovian networks [20, 18, 19, 17], where the network evolves as follows: starting from an arbitrary initial graph $\mathcal{G}_0 = (\mathcal{V}, \mathcal{E}_0)$, for $t \geq 1$, $\mathcal{G}_t = (\mathcal{V}, \mathcal{E}_t)$ is stochastically determined as

$$\mathbb{P}[\{v,w\} \in \mathcal{E}_t | \{v,w\} \notin \mathcal{E}_{t-1}] = p, \quad \mathbb{P}[\{v,w\} \notin \mathcal{E}_t | \{v,w\} \in \mathcal{E}_{t-1}] = q. \tag{10}$$

p and q are often referred to as edge birth rate and edge death rate, respectively. When $0 < p, q < 1$, it is well-known that, regardless of the initial graph $\mathcal{G}_0$, the process converges to (stationary) Erdős-Rényi graph $\mathcal{G}\left(\mathcal{V}, \frac{p}{p+q}\right)$. For our experiments, we start from the baseline ER graph and set $(p, q) = (0.01, 0.1)$. We plot an example trajectory in Figure 5a.

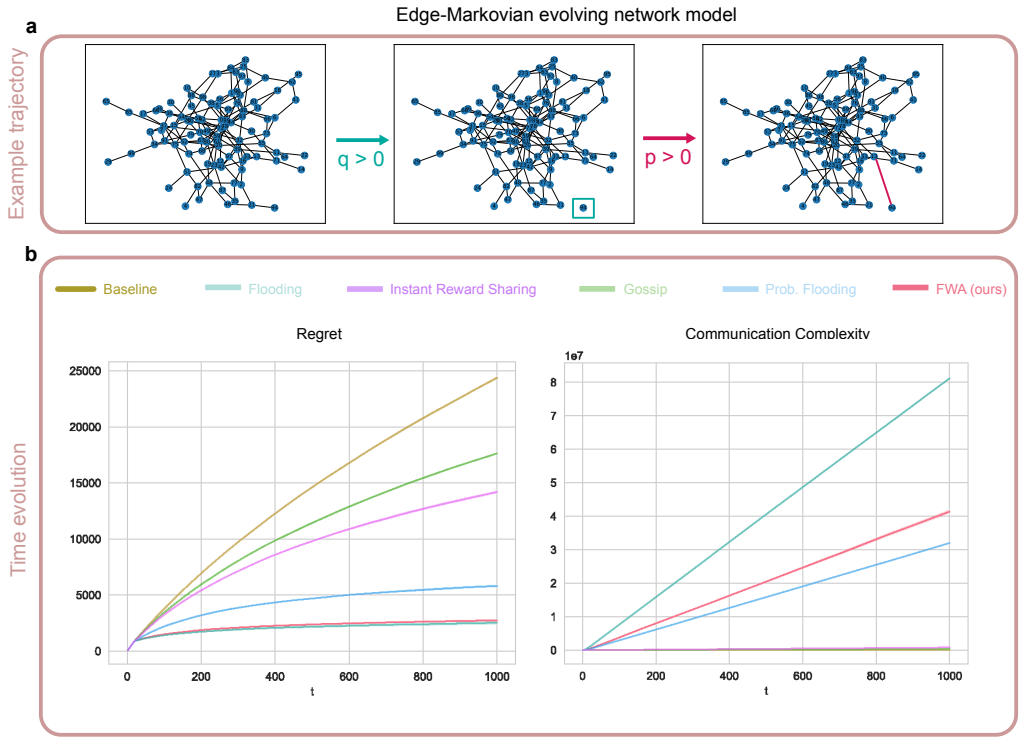


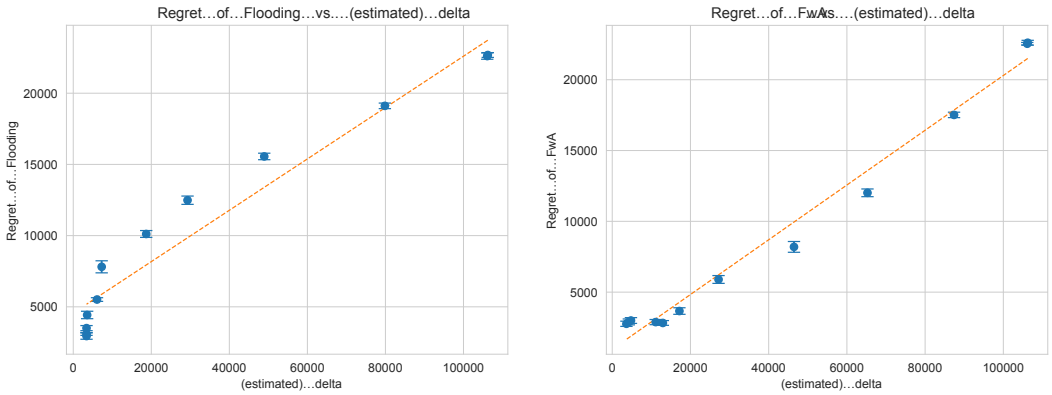
Figure 5 Comparing group regret and (cumulative) communication complexity in a dynamic network setting. Note that FWA achieves the same regret as FLOODING, with much lesser cumulative communication complexity.

The results are shown in Figure 5b. Observe how well our FWA protocol matches the regret of FLOODING, with *strictly* better communication complexity. This trend is consistent across all considered networks, showing that FWA is the most effective out of the considered protocols in dynamic networks. Similarly to the static case, we experimented with PROB. FLOODING of other stopping probabilities (not shown here) and observed that they tend to show worse trade-offs between regret and communication complexity. This suggests that the arm-dependent absorption mechanism of FWA implicitly regularizes the communication complexity in an efficient manner in dynamic networks, while minimizing the loss in regret.

6.4 Tightness of Theoretical Regret Upper Bounds

Recall that the theoretically derived regret bounds of FLOODING and FWA (Theorem 3.2 and 4.5) depend on both network topology and arm distribution. FLOODING scales with $\delta^{Flooding} := \sum_{\substack{a \in \mathcal{K} \\ \tilde{\Delta}_a > 0}} \frac{\theta([\mathcal{G}^\gamma]_{-a})}{\tilde{\Delta}_a}$ and FWA scales with $\delta^{FwA} := \sum_{\substack{a \in \mathcal{K} \\ \tilde{\Delta}_a > 0}} \frac{\theta\left([\mathcal{G}^\gamma_{(a,\mathbf{c})}]_{-a}\right)}{\tilde{\Delta}_a}$. To show that the theoretical bounds are tight and match well with practice, we perform an ablation study by varying the underlying edge density and seeing whether the aforementioned quantities scale well with the actual regrets.

We consider $\mathcal{G} \sim G(n, p)$ of varying edge density p and compare estimated δ's and regrets of FLOODING and FWA under the same setting as in previous experiments. Computing $\theta(\cdot)$'s requires computing the chromatic numbers, which is NP-hard. We thus approximate it with the size of a greedy coloring of the considered graph; for Erdős-Rényi graphs, the greedy coloring asymptotically results in twice the true chromatic number [25].



(a) Regret of FLOODING vs. $\delta^{Flooding}$. Orange line is the best linear fit ($R^2 = 0.9439$).

(b) Regret of FWA vs. δ^{FwA}. Orange line is the best linear fit ($R^2 = 0.9813$).

Figure 6 Experimental Results on δ. Note that there is a strong linear correlation between the estimated $\delta^{Flooding}, \delta^{FwA}$ and the final resulting regrets of FLOODING, FWA, respectively.

In Figure 6, we scatter plotted (δ, Regret) along with a best linear fit for FLOODING and FWA. Indeed, it can be seen that the relationship is almost linear, with high R^2, showing that our regret bounds indeed reflect the regrets in practice. There are some deviations from linearity, which we believe is due to small horizon length T and inaccuracy in estimating δ.

7 Conclusion

In this work, we described a novel setting for distributed multi-armed bandits, where agents communicate on an underlying network and do not all share the same arm set. We assume that each agent runs a UCB algorithm to identify their local best arm, and communicates the information they receive to their neighbors to minimize cumulative group regret. To deal with the very large communication complexity that however arises from using FLOODING in our setting, we then introduced a new communication protocol for complex networks, FLOODING WITH ABSORPTION (FWA). With FWA, agents forward information only if it pertains to an arm they themselves do not include in their arm set, whereas they absorb a message that gives information about one of their own arms. We provided theoretical upper and lower regret bounds and showed experimentally that FWA incurs only minimal group regret performance loss compared to FLOODING and even PROBABILISTIC FLOODING, while leading to a significantly improved communication complexity. In particular, we showed that FWA can reduce link congestion, which significantly improves upon simple heuristics such as probabilistic flooding. Our protocol is fully network-agnostic and hence does not need any fine-tuning, while still making use of the inherent heterogeneity of the problem instance. This makes it a very suitable choice for dynamically changing networks, or those that suffer from occasional message loss. We believe that our work highlights the importance of integrating network topology and action heterogeneity in the design of distributed bandit algorithms, and provides an efficient way to connect bandit learning and network protocols.

References

1 Animashree Anandkumar, Nithin Michael, and Ao Tang. Opportunistic Spectrum Access with Multiple Users: Learning under Competition. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010. doi:10.1109/INFCOM.2010.5462144.

2 Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2):235–256, 2002. doi:10.1023/A:1013689704352.

3 Orly Avner and Shie Mannor. Multi-user lax communications: A multi-armed bandit approach. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016. doi:10.1109/INFOCOM.2016.7524557.

4 Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999. doi:10.1126/science.286.5439.509.

5 Albert-László Barabási and Réka Albert. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, 2002. doi:10.1103/RevModPhys.74.47.

6 Oded Berger-Tal, Jonathan Nathan, Ehud Meron, and David Saltz. The Exploration-Exploitation Dilemma: A Multidisciplinary Framework. *PLOS ONE*, 9:1–8, apr 2014. doi:10.1371/journal.pone.0095693.

7 Béla Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer, 2002. doi:10.1007/978-1-4612-0619-4.

8 Sébastien Bubeck. *Bandits Games and Clustering Foundations*. PhD thesis, INRIA Nord Europe, jun 2010.

9 Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012. doi:10.1561/2200000024.

10 Swapna Buccapatnam, Jian Tan, and Li Zhang. Information sharing in distributed stochastic bandits. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2605–2613, 2015. doi:10.1109/INFOCOM.2015.7218651.

11 Keren Censor-Hillel, Bernhard Haeupler, Jonathan Kelner, and Petar Maymounkov. Global Computation in a Poorly Connected World: Fast Rumor Spreading with No Dependence on Conductance. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 961–970. Association for Computing Machinery, 2012. doi:10.1145/2213977.2214064.

12 Nicolò Cesa-Bianchi, Tommaso Cesari, and Claire Monteleoni. Cooperative Online Learning: Keeping your Neighbors Updated. In *Proceedings of the 31st International Conference on Algorithmic Learning Theory*, volume 117 of *Proceedings of Machine Learning Research*, pages 234–250. PMLR, 08 February–11 February 2020. URL: http://proceedings.mlr.press/v117/cesa-bianchi20a.html.

13 Nicholas B Chang and Mingyan Liu. Controlled Flooding Search in a Large Network. *IEEE/ACM Transactions on Networking*, 15(2):436–449, 2007. doi:10.1145/1279660.1279675.

14 Ronshee Chawla, Abishek Sankararaman, Ayalvadi Ganesh, and Sanjay Shakkottai. The Gossiping Insert-Eliminate Algorithm for Multi-Agent Bandits. In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 3471–3481. PMLR, 26–28 August 2020. URL: http://proceedings.mlr.press/v108/chawla20a.html.

15 Ronshee Chawla, Daniel Vial, Sanjay Shakkottai, and R. Srikant. Collaborative Multi-Agent Heterogeneous Multi-Armed Bandits. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 4189–4217. PMLR, 23–29 July 2023. URL: https://proceedings.mlr.press/v202/chawla23a.html.

16 Yu-Zhen Janice Chen, Lin Yang, Xuchuang Wang, Xutong Liu, Mohammad Hajiesmaili, John C. S. Lui, and Don Towsley. On-Demand Communication for Asynchronous Multi-Agent Bandits. In *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pages 3903–3930. PMLR, 25–27 April 2023. URL: https://proceedings.mlr.press/v206/chen23c.html.

17 Andrea Clementi, Pierluigi Crescenzi, Carola Doerr, Pierre Fraigniaud, Francesco Pasquale, and Riccardo Silvestri. Rumor Spreading in Random Evolving Graphs. *Random Structures & Algorithms*, 48(2):290–312, 2016. doi:10.1002/RSA.20586.

18 Andrea Clementi, Angelo Monti, Francesco Pasquale, and Riccardo Silvestri. Information Spreading in Stationary Markovian Evolving Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 22(9):1425–1432, 2011. doi:10.1109/TPDS.2011.33.

19 Andrea Clementi, Riccardo Silvestri, and Luca Trevisan. Information spreading in dynamic graphs. *Distributed Computing*, 28(1):55–73, feb 2015. doi:10.1007/S00446-014-0219-2.

20 Andrea E. F. Clementi, Claudio Macci, Angelo Monti, Francesco Pasquale, and Riccardo Silvestri. Flooding Time of Edge-Markovian Evolving Graphs. *SIAM Journal on Discrete Mathematics*, 24(4):1694–1712, 2010. doi:10.1137/090756053.

21 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 4 edition, 2022.

22 Abhimanyu Dubey and Alex Pentland. Cooperative Multi-Agent Bandits with Heavy Tails. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 2730–2739. PMLR, 13–18 July 2020. URL: http://proceedings.mlr.press/v119/dubey20a.html.

23 Paul Erdős and Alfréd Rényi. On random graphs, I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

24 P.T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. Epidemic Information Dissemination in Distributed Systems. *Computer*, 37(5):60–67, 2004. doi:10.1109/MC.2004.1297243.

25 Alan Frieze and Colin McDiarmid. Algorithmic Theory of Random Graphs. *Random Structures & Algorithms*, 10(1–2):5–42, feb 1997. doi:10.1002/(SICI)1098-2418(199701/03)10:1/2\%3C5::AID-RSA2\%3E3.0.CO;2-Z.

26 E. N. Gilbert. Random Graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

27 A. Gyárfás. Problems from the world surrounding perfect graphs. *Applicationes Mathematicae*, 19(3-4):413–441, 1987.

28 András Gyárfás, András Sebő, and Nicolas Trotignon. The chromatic gap and its extremes. *Journal of Combinatorial Theory, Series B*, 102(5):1155–1178, 2012. doi:10.1016/J.JCTB.2012.06.001.

29 Bernhard Haeupler. Simple, Fast and Deterministic Gossip and Rumor Spreading. *Journal of the ACM*, 62(6), dec 2015. doi:10.1145/2767126.

30 Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, 2008.

31 Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137, 1983. doi:10.1016/0378-8733(83)90021-7.

32 Long Jin, Shuai Li, Lin Xiao, Rongbo Lu, and Bolin Liao. Cooperative Motion Generation in a Distributed Network of Redundant Robot Manipulators With Noises. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(10):1715–1724, 2018. doi:10.1109/TSMC.2017.2693400.

33 Andrea Kadović, Sebastian M. Krause, Guido Caldarelli, and Vinko Zlatic. Bond and site color-avoiding percolation in scale-free networks. *Physical Review E*, 98:062308, dec 2018. doi:10.1103/PhysRevE.98.062308.

34 Ravi Kumar Kolla, Krishna Jagannathan, and Aditya Gopalan. Collaborative Learning of Stochastic Bandits Over a Social Network. *IEEE/ACM Transactions on Networking*, 26(4):1782–1795, 2018. doi:10.1109/TNET.2018.2852361.

35 Sebastian M. Krause, Michael M. Danziger, and Vinko Zlatić. Hidden Connectivity in Networks with Vulnerable Classes of Nodes. *Phys. Rev. X*, 6:041022, oct 2016. doi:10.1103/PhysRevX.6.041022.

36 Sebastian M. Krause, Michael M. Danziger, and Vinko Zlatić. Color-avoiding percolation. *Physical Review E*, 96:022313, aug 2017. doi:10.1103/PhysRevE.96.022313.

37 T.L Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985. doi:10.1016/0196-8858(85)90002-8.

38 Peter Landgren, Vaibhav Srivastava, and Naomi Ehrich Leonard. Distributed cooperative decision making in multi-agent multi-armed bandits. *Automatica*, 125:109445, 2021. doi:10.1016/J.AUTOMATICA.2020.109445.

39 Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020. doi:10.1017/9781108571401.

40 Marc Lelarge, Alexandre Proutière, and M. Sadegh Talebi. Spectrum bandit optimization. In *2013 IEEE Information Theory Workshop (ITW)*, pages 1–5, 2013. doi:10.1109/ITW.2013.6691221.

41 Daniel A Levinthal and James G March. The myopia of learning. *Strategic management journal*, 14(S2):95–112, 1993. doi:10.1002/smj.4250141009.

42 Feng Li, Dongxiao Yu, Huan Yang, Jiguo Yu, Holger Karl, and Xiuzhen Cheng. Multi-Armed-Bandit-Based Spectrum Scheduling Algorithms in Wireless Networks: A Survey. *IEEE Wireless Communications*, 27(1):24–30, 2020. doi:10.1109/MWC.001.1900280.

43 Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A Contextual-Bandit Approach to Personalized News Article Recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 661–670. Association for Computing Machinery, 2010. doi:10.1145/1772690.1772758.

44 Shuai Li, Ruofan Kong, and Yi Guo. Cooperative Distributed Source Seeking by Multiple Robots: Algorithms and Experiments. *IEEE/ASME Transactions on Mechatronics*, 19(6):1810–1820, 2014. doi:10.1109/TMECH.2013.2295036.

45 H Lim and C Kim. Flooding in wireless ad hoc networks. *Computer Communications*, 24(3):353–363, 2001. doi:10.1016/S0140-3664(00)00233-4.

46 Keqin Liu and Qing Zhao. Distributed Learning in Multi-Armed Bandit With Multiple Players. *IEEE Transactions on Signal Processing*, 58(11):5667–5681, 2010. doi:10.1109/TSP.2010.2062509.

47 Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 84–95, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/514191.514206.

48 Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can Heterogeneity Make Gnutella Scalable? In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 94–103, Berlin, Heidelberg, 2002. Springer-Verlag. doi:10.1007/3-540-45748-8_9.

49 Udari Madhushani, Abhimanyu Dubey, Naomi Leonard, and Alex Pentland. One More Step Towards Reality: Cooperative Bandits with Imperfect Communication. In *Advances in Neural Information Processing Systems*, volume 34, pages 7813–7824. Curran Associates, Inc., 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/40cb228987243c91b2dd0b7c9c4a0856-Abstract.html.

50 Udari Madhushani and Naomi Ehrich Leonard. Distributed Bandits: Probabilistic Communication on d-regular Graphs. In *2021 European Control Conference (ECC)*, pages 830–835, 2021. doi:10.23919/ECC54610.2021.9655031.

51 James G. March. Exploration and Exploitation in Organizational Learning. *Organization Science*, 2(1):71–87, 1991. doi:10.1287/orsc.2.1.71.

52 Katja Mehlhorn, Ben R Newell, Peter M Todd, Michael D Lee, Kate Morgan, Victoria A Braithwaite, Daniel Hausmann, Klaus Fiedler, and Cleotilde Gonzalez. Unpacking the exploration–exploitation tradeoff: A synthesis of human and animal literatures. *Decision*, 2(3):191–215, 2015. doi:10.1037/dec0000033.

53 Andrea Munaro. Bounded clique cover of some sparse graphs. *Discrete Mathematics*, 340(9):2208–2216, 2017. doi:10.1016/J.DISC.2017.04.004.

54 Konstantinos Oikonomou, George Koufoudakis, Sonia Aïssa, and Ioannis Stavrakakis. Probabilistic Flooding Performance Analysis Exploiting Graph Spectra Properties. *IEEE/ACM Transactions on Networking*, 31(1):133–146, 2023. doi:10.1109/TNET.2022.3192310.

55 Ashikur Rahman, Wlodek Olesinski, and Pawel Gburzynski. Controlled flooding in wireless ad-hoc networks. In *International Workshop on Wireless Ad-Hoc Networks, 2004.*, pages 73–78. IEEE, 2004. doi:10.1109/IWWAN.2004.1525544.

56 Abishek Sankararaman, Ayalvadi Ganesh, and Sanjay Shakkottai. Social Learning in Multi Agent Multi Armed Bandits. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3), dec 2019. doi:10.1145/3366701.

57 Devavrat Shah. Gossip Algorithms. *Foundations and Trends® in Networking*, 3(1):1–125, 2009. doi:10.1561/1300000014.

58 K. Sugawara, T. Kazama, and T. Watanabe. Foraging behavior of interacting robots with virtual pheromone. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 3074–3079 vol.3, 2004. doi:10.1109/IROS.2004.1389878.

59 Balazs Szorenyi, Robert Busa-Fekete, Istvan Hegedus, Robert Ormandi, Mark Jelasity, and Balazs Kegl. Gossip-based distributed stochastic bandit algorithms. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) of *Proceedings of Machine Learning Research*, pages 19–27. PMLR, 17–19 June 2013. URL: http://proceedings.mlr.press/v28/szorenyi13.html.

60 Andrew S. Tanenbaum, Nick Feamster, and David Wetherall. *Computer Networks*. Pearson, 6 edition, 2021.

61 Yu-Chee Tseng, Sze-Yao Ni, Yuh-Shyan Chen, and Jang-Ping Sheu. The Broadcast Storm Problem in a Mobile Ad Hoc Network. *Wireless Networks*, 8(2):153–167, mar 2002. doi:10.1023/A:1013763825347.

62 Amin Vahdat and David Becker. Epidemic Routing for Partially-Connected Ad Hoc Networks. Technical Report CS-200006, Duke University, 2000.

63 Daniel Vial, Sanjay Shakkottai, and R. Srikant. Robust Multi-Agent Multi-Armed Bandits. In *Proceedings of the Twenty-Second International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, MobiHoc '21, pages 161–170. Association for Computing Machinery, 2021. doi:10.1145/3466772.3467045.

64 Milan Vojnović and Alexandre Proutière. Hop limited flooding over dynamic networks. In *2011 Proceedings IEEE INFOCOM*, pages 685–693, 2011. doi:10.1109/INFCOM.2011.5935249.

65 Po-An Wang, Alexandre Proutière, Kaito Ariu, Yassir Jedra, and Alessio Russo. Optimal Algorithms for Multiplayer Multi-Armed Bandits. In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 4120–4129. PMLR, 26–28 August 2020. URL: http://proceedings.mlr.press/v108/wang20m.html.

66 N. Wisitpongphan, O.K. Tonguz, J.S. Parikh, P. Mudalige, F. Bai, and V. Sadekar. Broadcast storm mitigation techniques in vehicular ad hoc networks. *IEEE Wireless Communications*, 14(6):84–94, 2007. doi:10.1109/MWC.2007.4407231.

67 Wenchao Xia, Tony Q. S. Quek, Kun Guo, Wanli Wen, Howard H. Yang, and Hongbo Zhu. Multi-Armed Bandit-Based Client Scheduling for Federated Learning. *IEEE Transactions on Wireless Communications*, 19(11):7108–7123, 2020. doi:10.1109/TWC.2020.3008091.

68 Lin Yang, Yu-Zhen Janice Chen, Mohammad H. Hajiemaili, John C. S. Lui, and Don Towsley. Distributed Bandits with Heterogeneous Agents. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 200–209, 2022. doi:10.1109/INFOCOM48880.2022.9796901.

69 Lin Yang, Yu-Zhen Janice Chen, Stephen Pasteris, Mohammad H. Hajiesmaili, John C. S. Lui, and Don Towsley. Cooperative Stochastic Bandits with Asynchronous Agents and Constrained Feedback. In *Advances in Neural Information Processing Systems*, volume 34, pages 8885–8897. Curran Associates, Inc., 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/4a5876b450b45371f6cfe5047ac8cd45-Abstract.html.

A Notations

Notations	
$\mathcal{G}$	Communication network
$\mathcal{V}$	Set of vertices in the graph, i.e., agents
$\mathcal{G}[S]$	Induced subgraph over $S \subset \mathcal{V}$
$N = \lvert\mathcal{V}\rvert$	The number of agents
$\mathcal{V}_a$	Set of agents having arm a
$\mathcal{V}_{-a}$	Set of agents having arm a as the suboptimal arm
$\mathcal{K}$	The set of all arms in the network
$\mathcal{K}_v$	The set of arms agent v has access to
$\mathcal{K}_{-v}$	The set of (locally) suboptimal arms agent v has access to
a^v_*	The best local arm for agent v
$\mu_{a^v_*}$	Reward of best local arm for agent v
$\hat{\mu}^v_a(t)$	Average reward that agent v has computed for arm a at time t
$M^v_a(t)$	Number of observations of arm a available to agent v at time t
$N^v_a(t)$	Number of times agent v pulls arm a up to time t
$N_a(t)$	Number of times all agents pull arm a up to time t
γ	Time-to-live of a message sent by an agent
$\mathcal{G}^\gamma$	Graph power of $\mathcal{G}$ of γ-th order: graph over $\mathcal{V}$ such that when $d_{\mathcal{G}}(v,w) \le \gamma$, $\{v,w\}$ is an edge
T	Time horizon
$R(T)$	Cumulative group regret over the time horizon
Δ^v_a	Agent-specific suboptimality gap

B Proof of Theorem 3.2 – Regret Upper Bound

The basic proof idea is to consider $\mathcal{G}^\gamma$, which is equivalent to the collaboration network via flooding with TTL γ. Then we consider a clique covering of $\mathcal{G}^\gamma$, and upper bound the group regret of the full collaboration by that in which collaboration happens only intra-clique. The deterministic delays between agents in each clique are precisely their distance, measured in the *original* communication network $\mathcal{G}$. We proceed similarly to [68] while filling in some missing details.

For each $a \in \mathcal{K}$ and $v \in V_{-a}$ (i.e., v contains a as a *(locally) suboptimal* arm), define $B^v_a(t) := \{a^v(t) = a\}$, which can be rewritten as

$$B^v_a(t) = \left\{ \hat{\mu}^v_{a'}(t) + \sqrt{\frac{\alpha \log t}{M^v_{a'}(t)}} < \hat{\mu}^v_a(t) + \sqrt{\frac{\alpha \log t}{M^v_a(t)}} \quad \forall a' \in \mathcal{K}_v \setminus \{a\} \right\}.$$

Thus, this implies that the following event holds:

$$E^v_a(t) := \left\{ \hat{\mu}^v_{a^v_\star}(t) + \sqrt{\frac{\alpha \log t}{M^v_{a^v_\star}(t)}} < \hat{\mu}^v_a(t) + \sqrt{\frac{\alpha \log t}{M^v_a(t)}} \right\}.$$

i.e. $B^v_a(t) \subseteq E^v_a(t)$.

Next, for $\ell^v_a := \frac{4\alpha \log T}{(\Delta^v_a)^2}$, define

$$\tau^v_a := \min\left\{t = 1, \cdots, T \ : \ M^v_a(t) > \ell^v_a\right\}, \tag{11}$$

i.e., τ^v_a is the first time at which agent v, and its neighbors, observe arm a at least ℓ^v_a times.

Denoting $c_a^v(t) = \sqrt{\frac{\alpha \log t}{M_a^v(t)}}$, we have that [8]

$$E_a^v(t) \subseteq \underbrace{\left\{\hat{\mu}_{a_\star^v}^v(t) + c_{a_\star^v}^v(t) \le \mu_{a_\star^v}\right\}}_{=A(t)} \cup \underbrace{\{\hat{\mu}_a^v(t) \ge \mu_a + c_a^v(t)\}}_{=B(t)} \cup \{M_a^v(t) < \ell_a^v\}.$$

Then the regret can be rewritten as follows [8]:

$$R(T) = \mathbb{E}\left[\sum_{a\in\mathcal{K}}\sum_{v\in V_a} \Delta_a^v N_a^v(T)\right] \le \underbrace{\sum_{a\in\mathcal{K}}\sum_{v\in V_a} \Delta_a^v \mathbb{E}[N_a^v(\tau_a^v)]}_{(a)} + \underbrace{\sum_{a\in\mathcal{K}}\sum_{v\in V_a} \Delta_a^v \sum_{t=1}^{T} \mathbb{P}[E_a^v(t) \cap \{t > \tau_a^v\}]}_{(b)}.$$

(a) is bounded using clique covering and Abel transformation:

▶ **Lemma B.1.** *For each $a \in \mathcal{K}$ with $\mathcal{V}_{-a} \neq \emptyset$, we have that*

$$\sum_{v\in V_a} \Delta_a^v \mathbb{E}[N_a^v(\tau_a^v)] \le \frac{4\alpha \log T}{\Delta_a^\gamma} + f_a(\gamma).$$

(b) is bounded as follows:

▶ **Lemma B.2.** *The following hold for each $v \in V$ and locally suboptimal $a \in \mathcal{K}_v$:*

$$\sum_{t=1}^{T} \mathbb{P}[E_t^v(a) \cap \{t > \tau_a^v\}] \le \left(\frac{\alpha + 1/2}{\alpha - 1/2}\right)^2 \frac{8(\gamma+1)}{\log \frac{(\gamma+1)(\alpha+1/2)}{4\sigma^2}}.$$

Combining both lemmas gives the desired statement.

B.1 Proof of Lemma B.1

Let $\mathcal{C}_{-a}$ be a (vertex-disjoint) clique cover of $[\mathcal{G}^\gamma]_{-a} = \mathcal{G}^\gamma[\mathcal{V}_{-a}]$, and let $C \in \mathcal{C}_{-a}$. Let $C = \{v_1, \cdots, v_c\}$ be such that $\Delta_a^{v_1} \ge \cdots \ge \Delta_a^{v_k} > 0$.

We fist have that $\mathbb{E}[N_a^{v_1}(\tau_a^{v_1})] \le \mathbb{E}[M_a^{v_1}(\tau_a^{v_1})] \le \frac{4\alpha \log T}{(\Delta_a^1)^2}$. For each $k \in \{2, \cdots, c\}$,

$$\begin{aligned}
\sum_{j=1}^{k} \mathbb{E}[N_a^{v_j}(\tau_a^{v_j})] &\le \mathbb{E}[N_a^{v_k}(\tau_a^{v_k})] + \sum_{j=1}^{k-1} \mathbb{E}[N_a^{v_j}(\tau_a^{v_j})] \\
&\overset{(a)}{\le} \mathbb{E}[M_a^{v_k}(\tau_a^{v_k})] + \sum_{j=1}^{k-1}\sum_{\iota=1}^{\gamma} \mathbb{1}[d_{\mathcal{G}}(v_k, v_j) = \iota] \left(\mathbb{E}[N_a^{v_j}(\tau_a^{v_j})] - \mathbb{E}[N_a^{v_j}(\tau_a^{v_k} - \iota)]\right) \\
&\overset{(b)}{\le} \frac{4\alpha \log T}{(\Delta_a^{v_k})^2} + \sum_{j=1}^{k-1}\sum_{\iota=1}^{\gamma} \mathbb{1}[d_{\mathcal{G}}(v_k, v_j) = \iota] \min\left(\gamma + \iota, \frac{4\alpha \log T}{(\Delta_a^{v_j})^2}\right) \\
&\le \frac{4\alpha \log T}{(\Delta_a^{v_k})^2} + \sum_{j=1}^{k-1} \min\left(2\gamma, \frac{4\alpha \log T}{(\Delta_a^{v_j})^2}\right).
\end{aligned}$$

Here, (a) follows from the simple decomposition of $M_a^{v_k}(\tau_a^{v_k})$:

$$\begin{aligned}
M_a^{v_k}(\tau_a^{v_k}) &= N_a^{v_k}(\tau_a^{v_k}) + \sum_{w\in\mathcal{V}_a\setminus\{v_k\}} \sum_{\iota=1}^{\gamma} \mathbb{1}[d_{\mathcal{G}}(v_k, w) = \iota] N_a^w(\tau_a^{v_k} - \iota) \\
&\ge N_a^{v_k}(\tau_a^{v_k}) + \sum_{j=1}^{k-1}\sum_{\iota=1}^{\gamma} \mathbb{1}[d_{\mathcal{G}}(v_k, v_j) = \iota] N_a^{v_j}(\tau_a^{v_k} - \iota).
\end{aligned}$$

(b) follows from the following reasoning: first, we have that $\tau_a^{v_k} + \gamma \geq \tau_a^{v_j}$ as at time $\tau_a^{v_k} + \gamma$, any remaining agent v_j should have at least $\ell_a^{v_k} \geq \ell_a^{v_j}$ observations of a, i.e., $\tau_a^{v_j}$ is already reached. Thus, we have that

$$\mathbb{E}[N_a^{v_j}(\tau_a^{v_j})] - \mathbb{E}[N_a^{v_j}(\tau_a^{v_k} - \iota)] \leq \mathbb{E}[N_a^{v_j}(\tau_a^{v_j})] \leq \frac{4\alpha \log T}{(\Delta_a^{v_j})^2},$$

and

$$\mathbb{E}[N_a^{v_j}(\tau_a^{v_j})] - \mathbb{E}[N_a^{v_k}(\tau_a^{v_k} - \iota)] \leq \mathbb{E}[N_a^{v_j}(\tau_a^{v_k} + \gamma)] - \mathbb{E}[N_a^{v_j}(\tau_a^{v_k} - \iota)] \leq \iota + \gamma.$$

Recall that the group regret for our clique C is $\sum_{k=1}^{c} \Delta_a^k \mathbb{E}[N_a^k(\tau_a^k)]$, where for simplicity we denote $\Delta_a^k := \Delta_a^{v_k}$. The important observation is that to make the worst-case regret upper-bound, we must "allocate" the most number of pulls to the arms with the largest gap, i.e.,

$$\begin{aligned}
&\sum_{k=1}^{c} \Delta_a^k \mathbb{E}[N_a^k(\tau_a^k)] \\
&\leq \Delta_a^1 \left(\frac{4\alpha \log T}{(\Delta_a^1)^2} \right) + \sum_{k=2}^{c} \Delta_a^k \left\{ \left(\frac{4\alpha \log T}{(\Delta_a^k)^2} - \frac{4\alpha \log T}{(\Delta_a^{k-1})^2} \right) + \min\left(2\gamma, \frac{4\alpha \log T}{(\Delta_a^{k-1})^2} \right) \right\} \\
&\overset{(*)}{=} \frac{4\alpha \log T}{\Delta_a^1} + \left(\Delta_a^c \frac{4\alpha \log T}{(\Delta_a^c)^2} - \Delta_a^1 \frac{4\alpha \log T}{(\Delta_a^1)^2} \right) + \sum_{k=1}^{c-1} \frac{4\alpha \log T}{(\Delta_a^k)^2} (\Delta_a^k - \Delta_a^{k+1}) + \sum_{k=2}^{c} \Delta_a^k \min\left(2\gamma, \frac{4\alpha \log T}{(\Delta_a^{k-1})^2} \right) \\
&\leq \frac{4\alpha \log T}{\Delta_a^c} + \int_{\Delta_a^c}^{\Delta_a^1} \frac{4\alpha \log T}{z^2} dz + \sum_{k=2}^{c+1} \Delta_a^{k-1} \min\left(2\gamma, \frac{4\alpha \log T}{(\Delta_a^{k-1})^2} \right) \\
&= \frac{8\alpha \log T}{\Delta_a^c} - \frac{4\alpha \log T}{\Delta_a^1} + \sum_{k=1}^{c} \Delta_a^k \min\left(2\gamma, \frac{4\alpha \log T}{(\Delta_a^k)^2} \right) \\
&= 4\alpha \log T \left(\frac{2}{\min_{v \in C} \Delta_a^v} - \frac{1}{\max_{v \in C} \Delta_a^v} \right) + \sum_{k=1}^{c} \Delta_a^k \min\left(2\gamma, \frac{4\alpha \log T}{(\Delta_a^k)^2} \right),
\end{aligned}$$

where $(*)$ follows from the Abel transformation.

Thus,

$$\begin{aligned}
&\sum_{v \in V_a} \Delta_a^v \mathbb{E}[N_a^v(\tau_a^v)] \\
&= \min_{\mathcal{C}_{-a}} \sum_{C \in \mathcal{C}_{-a}} \sum_{v \in C} \Delta_a^v \mathbb{E}[N_a^v(\tau_a^v)] \\
&\leq \min_{\mathcal{C}_{-a}} \sum_{C \in \mathcal{C}_{-a}} \left\{ 4\alpha \log T \left(\frac{2}{\min_{v \in C} \Delta_a^v} - \frac{1}{\max_{v \in C} \Delta_a^v} \right) \right\} + \sum_{v \in \mathcal{V}_{-a}} \Delta_a^v \min\left(2\gamma, \frac{4\alpha \log T}{(\Delta_a^v)^2} \right) \\
&= \frac{4\alpha \log T}{\Delta_a^\gamma} + f_a(\gamma).
\end{aligned}$$

B.2 Proof of Lemma B.2

We start by noting that

$$\mathbb{P}\left[E_a^v(a) \cap \{t > \tau_a^v\}\right] \leq \mathbb{P}\left[A(t) \cap \{t > \tau_a^v\}\right] + \mathbb{P}[B(t) \cap \{t > \tau_a^v\}].$$

We consider $\mathbb{P}\left[A(t) \cap \{t > \tau_a^v\}\right]$ first. For simplicity, denote $M_\star^v(t) := M_{a_\star^v}^v(t)$. The initialization phase of Algorithm 1 implies that

$$\sum_{\iota=0}^{\gamma} \left| \{w \in \mathcal{V}_a : d_{\mathcal{G}}(w, v) = \iota\} \right| \leq M_\star^v(t) \leq \sum_{\iota=0}^{\gamma} (t - \iota) \left| \{w \in \mathcal{V}_a : d_{\mathcal{G}}(w, v) = \iota\} \right|.$$

Denote $[a,b] := \{\lceil a \rceil, \lceil a \rceil + 1, \cdots, \lfloor b \rfloor\}$, $\mathcal{N}_\iota = \{w \in \mathcal{V}_a : d_{\mathcal{G}}(w,v) = \iota\}$, and $N_\iota := |\mathcal{N}_\iota|$. Also, with a slight abuse of notation, here let us denote $X^u_{a^v_\star}(k)$ to be the reward received by agent v when she pulls arm $a_\star$ for the k-th time. Then,

$$
\begin{aligned}
&\mathbb{P}\left[A(t) \cap \{t > \tau^v_a\}\right] \\
&\le \mathbb{P}\left[\exists \{s_\iota\}_{\iota\in[0,\gamma]}, s_\iota \in [N_\iota, (t-\iota)N_\iota] \;\; \exists \{s_\iota(u)\}_{u\in\mathcal{N}_\iota} : \sum_{u\in\mathcal{N}_\iota} s_\iota(u) = s_\iota, \sum_{\iota=0}^{\gamma} s_\iota = s \right. \\
&\qquad \left. \text{s.t.} \;\; \frac{1}{s}\sum_{\iota=0}^{\gamma}\sum_{u\in\mathcal{N}_\iota}\sum_{k=1}^{s_\iota(u)} X^u_{a^v_\star}(k) + \sqrt{\frac{\alpha\log t}{s}} \le \mu^v_\star \right] \\
&\le \mathbb{P}\left[\exists \{s_\iota\}_{\iota\in[0,\gamma]}, s_\iota \in [N_\iota, (t-\iota)N_\iota] \;\; \exists \{s_\iota(u)\}_{u\in\mathcal{N}_\iota} : \sum_{u\in\mathcal{N}_\iota} s_\iota(u) = s_\iota, \sum_{\iota=0}^{\gamma} s_\iota = s \right. \\
&\qquad \left. \text{s.t.} \;\; \sum_{\iota=0}^{\gamma}\sum_{u\in\mathcal{N}_\iota}\sum_{k=1}^{s_\iota(u)} \left(X^u_{a^v_\star}(k) - \mu^v_\star\right) \le -\sqrt{s\alpha\log t} \right] \\
&\le \sum_{\iota=0}^{\gamma} \mathbb{P}\left[\exists s_\iota \in [N_\iota, (t-\iota)N_\iota] \;\; \text{s.t.} \;\; \sum_{k=1}^{s_\iota} Y(k) \le -\sqrt{s_\iota(\gamma+1)\alpha\log t}\right],
\end{aligned}
$$

where $Y(k) \overset{d}{=} X^u_{a^v_\star}(k) - \mu^v_\star$ is *i.i.d.* σ-subGaussian random variable with $\mathbb{E}[Y(k)] = 0$.

Before moving forward, we recall a maximal-type concentration result:

▶ **Theorem B.3** (Theorem 9.2 of [39]). *Let $Y_1, \cdots, Y_n$ be a sequence of independent σ-subGaussian random variables, and let $S_t = \sum_{s=1}^t Y_s$. Then, for any $\varepsilon > 0$,*

$$
\mathbb{P}\left[\exists t \le n : S_t \ge \varepsilon\right] \le \exp\left(-\frac{\varepsilon^2}{2n\sigma^2}\right). \tag{12}
$$

Using the above concentration result as well as the peeling argument on a geometric grid [8, 34], we have that for any $\{\beta_\iota\}_{\iota\in[0,\gamma]}$ with $\beta_\iota \in \left(\frac{2\sigma^2}{(\gamma+1)\alpha}, 1\right)$,

$$
\begin{aligned}
&\mathbb{P}[A(t)] \\
&\le \sum_{\iota=0}^{\gamma}\sum_{j=0}^{\frac{\log(t-\iota)}{\log(1/\beta_\iota)}} \mathbb{P}\left[\exists s_\iota \in \left[N_\iota\beta_\iota^{j+1}(t-\iota), N_\iota\beta_\iota^{j}(t-\iota)\right] \;\; \text{s.t.} \;\; \sum_{k=1}^{s_\iota} Y(k) \le -\sqrt{s_\iota(\gamma+1)\alpha\log t}\right] \\
&\le \sum_{\iota=0}^{\gamma}\sum_{j=0}^{\frac{\log(t-\iota)}{\log(1/\beta_\iota)}} \mathbb{P}\left[\exists s_\iota \in \left[N_\iota\beta_\iota^{j+1}(t-\iota), N_\iota\beta_\iota^{j}(t-\iota)\right] \;\; \text{s.t.} \;\; \sum_{k=1}^{s_\iota} Y(k) \le -\sqrt{N_\iota\beta_\iota^{j+1}(t-\iota)(\gamma+1)\alpha\log t}\right] \\
&\le \sum_{\iota=0}^{\gamma}\sum_{j=0}^{\frac{\log(t-\iota)}{\log(1/\beta_\iota)}} \mathbb{P}\left[\exists s_\iota \in \left[1, N_\iota\beta_\iota^{j}(t-\iota)\right] \;\; \text{s.t.} \;\; \sum_{k=1}^{s_\iota} Y(k) \le -\sqrt{N_\iota\beta_\iota^{j+1}(t-\iota)(\gamma+1)\alpha\log t}\right] \\
&\le \sum_{\iota=0}^{\gamma}\sum_{j=0}^{\frac{\log(t-\iota)}{\log(1/\beta_\iota)}} \exp\left(-\frac{N_\iota\beta_\iota^{j+1}(t-\iota)(\gamma+1)\alpha\log t}{2\sigma^2 N_\iota\beta_\iota^{j}(t-\iota)}\right) \\
&= \sum_{\iota=0}^{\gamma}\left(1+\frac{\log(t-\iota)}{\log(1/\beta_\iota)}\right) t^{-\frac{\beta_\iota(\gamma+1)\alpha}{2\sigma^2}} \\
&\le \sum_{\iota=0}^{\gamma}\left(1+\frac{\log(t-\iota)}{\log(1/\beta_\iota)}\right) (t-\iota)^{-\frac{\beta_\iota(\gamma+1)\alpha}{2\sigma^2}}.
\end{aligned}
$$

One can show the same for $\mathbb{P}[B(t)]$. (Note how both of them do not depend on any graph theoretical quantities).

Then,

$$\begin{aligned}
\sum_{t=1}^{T}\mathbb{P}[E_t^v(a)\cap\{t>\tau_a^v\}] &\le 2\sum_{t=\tau_a^v+1}^{T}\sum_{\iota=0}^{\gamma}\left(1+\frac{\log(t-\iota)}{\log(1/\beta_\iota)}\right)(t-\iota)^{-\frac{\beta_\iota(\gamma+1)\alpha}{2\sigma^2}}\\
&\le 2\sum_{\iota=0}^{\gamma}\sum_{t=\iota+1}^{\infty}\left(1+\frac{\log(t-\iota)}{\log(1/\beta_\iota)}\right)(t-\iota)^{-\frac{\beta_\iota(\gamma+1)\alpha}{2\sigma^2}}\\
&\le 2\sum_{\iota=0}^{\gamma}\int_{\iota+1}^{\infty}\left(1+\frac{\log(t-\iota)}{\log(1/\beta_\iota)}\right)(t-\iota)^{-\frac{\beta_\iota(\gamma+1)\alpha}{2\sigma^2}}dt\\
&= 2\sum_{\iota=0}^{\gamma}\int_{1}^{\infty}\left(1+\frac{\log t}{\log(1/\beta_\iota)}\right)t^{-\frac{\beta_\iota(\gamma+1)\alpha}{2\sigma^2}}dt\\
&= \frac{8(\gamma+1)}{\left(\frac{\beta(\gamma+1)\alpha}{2\sigma^2}-1\right)^2\log(1/\beta)},
\end{aligned}$$

where we've set $\beta_\iota=\beta$ for all ι's.

Following [8], we choose $\beta=\frac{4\sigma^2}{\gamma+1}\frac{1}{\alpha+\frac{1}{2}}$. Then,

$$\sum_{t=1}^{T}\mathbb{P}[E_t^v(a)\cap\{t>\tau_a^v\}]\le\left(\frac{\alpha+1/2}{\alpha-1/2}\right)^2\frac{8(\gamma+1)}{\log\frac{(\gamma+1)(\alpha+1/2)}{4\sigma^2}}.$$

C Additional Discussions on the Theoretical Results

C.1 Main Technical Challenges in the Proof of Theorem 3.2 – Regret Upper Bound

One might ask whether it is possible to use a star decomposition-type argument for our setting similar to [34], which could give us an improvement from $\theta([\mathcal{G}^\gamma]_{-a})$ to $\alpha([\mathcal{G}^\gamma]_{-a})$. We believe this is *not* possible, and the reason is as follows.

We note that the main technical challenge is that unlike the homogeneous settings [50, 34], the agents and the arms are intertwined: In a homogeneous setting, one could rewrite the regret as

$$R(T)=\sum_{v\in V}\sum_{a\in\mathcal{K}}\Delta_a^{Kolla}\mathbb{E}[N_a^v(T)]=\sum_{a\in\mathcal{K}}\Delta_a^{Kolla}\sum_{v\in V}\mathbb{E}[N_a^v(T)],$$

i.e., the regret can be decomposed such that one only needs to upper bound the number of visitations of each agent v. Thus with a star decomposition, for each star, it can be easily seen that the number of visitations of the leaf agents is precisely that of the center agent, allowing for us to further decompose $\sum_{v\in V}\mathbb{E}[N_a^v(T)]$ to sum of $\mathbb{E}[M_a^{v^{center}}(T)]$ over all center agents v^{center}. This is at the core of dealing with more general homogeneous settings such as when there is a communication network [34], possibly with faults [50].

Such a decomposition is *not* possible when the agents are heterogeneous, as the maximal suboptimality gap of v, Δ_a^v, is agent-dependent. To deal with such heterogeneity in the full information sharing (fully-connected graph) setting, Yang et al. [68] first ordered the agents according to Δ_a^v, then bounded the *cumulative* number of times suboptimal arm a is visited by agents $v_1,\cdots,v_k\in\mathcal{V}_{-a}$ via the design of UCB algorithm. Then, based on the intuition that the worst-case regret bound occurs when the arm a with the highest Δ_a^v is visited at its

maximum, the final regret bound is derived via the Abel transformation. The fact that all agents have access to all other agents' information is crucial in this proof idea. This can be seen from a very simple example; consider a situation in which agent v_1 has a very difficult problem (very small $\Delta_a^{v_1}$) but has lots of connections, and agent v_2 has a somewhat easy problem but has few connections. In this case, from the collaboration, it may be that v_1 learns faster than v_2, which can impact the ordering of the agents, which in turn impacts the whole Abel transformation-based argument. In a sense, our proof combines these two ideas. We start with a clique decomposition, instead of the star decomposition, of the graph in order to upper-bound the group regret with the sum of regrets of each clique. Then, we apply the Abel transformation-type argument to each clique.

Lastly, we remark that our results can also be easily extended to the setting where the agents asynchronously pull the arms, i.e., each agent v pulls arms at every ω_v round, with $\omega_v \geq 1$. It would be an interesting future direction if we could further reduce the communication cost in the asynchronous case based on ideas from recent works [16, 69, 68].

C.2 Extending Theorem 5.2 – Regret Lower Bound

In order to match the lower bound to the upper bound in terms of graph topology and arm distribution-dependent constant as well, there are two avenues, both of which are inspired by Kolla et al [34], who consider the homogeneous setting with a general graph. On the one hand, it might be valuable to consider a Follow-Your-Leader-type policy, which would tighten the upper bound to match our lower bound. However, it is not clear how to choose the leaders in our heterogeneous setting, let alone how to ensure that followers get relevant information in terms of arms. Another way is to consider a more restricted class of policies, namely NAIC (non-altruistic & individually consistent) policies, which would tighten the lower bound to match our upper bound. Extending such notion to our setting of heterogeneous bandits over a graph *while* taking the communication protocol into account, e.g., whether we use the entire graph (FLOODING) or we use part of the graph (FWA), and seeing whether we can match the lower bound up to the derived upper bounds, is another interesting future direction. On a separate note, deriving a minimax lower bound, as done in Madhushani et al. [49], for our setting is also an interesting future direction.

Fault-Tolerant Computing with Unreliable Channels

Alejandro Naser-Pastoriza
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid, Spain

Gregory Chockler
University of Surrey, Guildford, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

We study implementations of basic fault-tolerant primitives, such as consensus and registers, in message-passing systems subject to process crashes and a broad range of communication failures. Our results characterize the necessary and sufficient conditions for implementing these primitives as a function of the connectivity constraints and synchrony assumptions. Our main contribution is a new algorithm for partially synchronous consensus that is resilient to process crashes and channel failures and is optimal in its connectivity requirements. In contrast to prior work, our algorithm assumes the most general model of message loss where faulty channels are *flaky*, i.e., can lose messages without any guarantee of fairness. This failure model is particularly challenging for consensus algorithms, as it rules out standard solutions based on leader oracles and failure detectors. To circumvent this limitation, we construct our solution using a new variant of the recently proposed *view synchronizer* abstraction, which we adapt to the crash-prone setting with flaky channels.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Consensus, network partitions, liveness, synchronizers

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.21

Related Version *Extended Version*: `https://arxiv.org/abs/2305.15150`

Funding This work was partially supported by the PRODIGY and DECO projects funded by MCIN/AEI, the BLOQUES project funded by the Madrid regional government, and by a research grant from Nomadic Labs.

1 Introduction

We are concerned with implementing basic fault-tolerant primitives, such as registers and consensus, in systems where processes can crash and communication channels may lose messages. This setting is of practical importance, since channel failures regularly occur in real-world deployments [10, 15, 33]. They arise from a variety of reasons – physical infrastructure failures, bugs in switch software, configuration errors – and they often lead to system outages. For example, Alquraan et al. [7] conducted a comprehensive study of failures due to network partitions in widely used replicated data stores. It found that a majority of such failures led to catastrophic effects and that the resolution of almost half of these failures required redesigning a system mechanism. Thus, the failures were not simply due to coding bugs, but to design flaws.

Perhaps the most challenging aspect of real-world network failures is that the ways the network can break down can be arbitrarily complex, resulting in a wide variety of connectivity configurations. While in the simplest case individual channel failures may not affect the overall network connectivity (Figure 1a), in more complex scenarios the network may become partitioned into multiple components, some of which may end up connected in

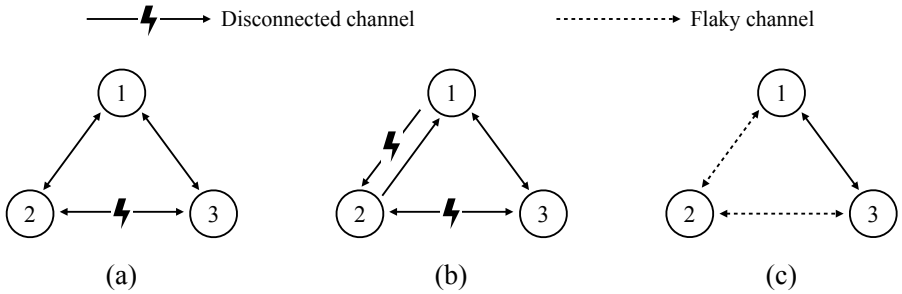


Figure 1 Examples of irregular connectivity configurations: (a) indirect connectivity; (b) asymmetric connectivity; and (c) flaky connectivity. All processes are correct.

one direction, but not in the other (Figure 1b). In the worst case, some components may become intermittently connected, causing an arbitrary subset of the messages transmitted between them to get lost (Figure 1c).

Intermittent connectivity has so far received little attention in the theory research. Most network models for studying consensus assume that channels are either (eventually) reliable or (eventually) disconnected (Figures 1a-b). Reliable channels are sometimes replaced by *fair lossy* ones, which only guarantee to deliver a message if it was sent infinitely many times. However, reliable and fair lossy channels are computationally equivalent: the former can be implemented from the latter by repeatedly resending each message until it gets acknowledged and filtering out duplicates [1].

Although the classical abstractions of failure or leader detectors [17] can be adapted to solve consensus in the above settings [4, 21, 22, 26, 35, 46], they are no longer useful in the presence of unconstrained message loss. Intuitively, the reason is that in this case they may fail to correctly identify processes with reliable connectivity, which is necessary to ensure the liveness of consensus. For example, suppose that the pattern of message loss experienced by the channels $2 \leftrightarrow 1$ and $2 \leftrightarrow 3$ in Figure 1c is such that all leader election messages are getting through. Thus, it is possible for process 2 to be elected as a leader. However, since any message sent by process 2 after being elected can be dropped (no matter how many times it is resent), the process may not be able to drive consensus to completion, violating liveness.

In this paper we investigate the possibility of implementing basic fault-tolerant abstractions resilient to a broad range of communication failures, including those induced by intermittent connectivity. To this end, we obtain several lower and upper bounds that characterize the solvability of registers and consensus as a function of the channel failure model, connectivity constraints and synchrony assumptions. Our main contribution is a new crash fault-tolerant algorithm that solves partially synchronous consensus under the *most general* model of message loss and is *optimal* in terms of its network connectivity requirements. Our algorithm furthermore offers a new way of constructing modular protocols for unreliable networks, which does not rely on the leader election or failure detection abstractions. Below we review our results in more detail.

Lower bounds. We first present a general framework for specifying fault-tolerance assumptions on process and channel failures (§2-3) by generalizing the classical notion of a *fail-prone system* [39]. For example, this framework allows us to specify whether an algorithm is meant to tolerate the failure patterns depicted in Figure 1. We then use this framework to establish minimal connectivity requirements necessary to implement registers or consensus (§4). To

Table 1 System models with different channel types and synchrony assumptions.

	Reliable/Disconnected (lower bound)	Eventually reliable/Flaky (upper bound)
Asynchronous (register)	$\mathcal{M}_{\text{ARD}}$	$\mathcal{M}_{\text{AEF}}$
Partially synchronous (consensus)	$\mathcal{M}_{\text{PRD}}$	$\mathcal{M}_{\text{PEF}}$

obtain strong lower bounds, we consider strong network models where correct channels are reliable and faulty channels are disconnected, i.e., drop all messages. We consider asynchrony for registers (model $\mathcal{M}_{\text{ARD}}$ in Table 1) and partial synchrony [23] for consensus (model $\mathcal{M}_{\text{PRD}}$): the execution starts with an asynchronous period and then becomes synchronous. The most interesting aspect of our lower bounds is that, unlike the prior work on consensus under unreliable connectivity (e.g., [2, 49]), they are proven for a very weak termination guarantee that only requires obstruction-free termination at a *subset* of processes. Informally, we show that for any n-process implementation of a register or consensus:

1. All processes where obstruction-freedom holds must be strongly connected via correct channels.
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 $\geq k + 1$ correct processes strongly connected by correct channels.

We call the largest set satisfying the condition in (2) the *connected core* of the network. For example, in Figure 1a the connected core is $\{1, 2, 3\}$, whereas in Figures 1b-c it is $\{1, 3\}$.

The above results generalize the celebrated CAP theorem [14, 27], which says that it is impossible to guarantee all of Consistency, Availability and network Partition-tolerance (§4.1). Whereas CAP only says that to ensure availability and consistency, *some* channels must be correct, we establish *how many* are needed; and whereas CAP requires availability at *all* processes, we establish conditions required to ensure it only in *a part* of the system. Result (2) furthermore establishes stronger requirements for algorithms resilient to a minority of crashes (which is optimal [23, 37]). It shows that, even if obstruction-freedom is required only at a *single* process, a majority thereof must still be strongly connected by correct channels.

Upper bounds. The second part of our contribution is to propose algorithms for registers and consensus that match our lower bounds. These algorithms assume the existence of the connected core and guarantee wait-freedom at all of its members. They are designed to work in an adversarial model where correct channels are only *eventually* reliable and faulty channels are *flaky*, i.e., can drop any messages sent on them without any guarantee of fairness (models $\mathcal{M}_{\text{AEF}}$ and $\mathcal{M}_{\text{PEF}}$ in Table 1). Flakiness is a very broad failure mode that captures a number of failure patterns occurring in practice, including both full and intermittent loss of connectivity. It also subsumes the previously considered variants of lossy channels, such as eventually disconnected [4, 18] and fair lossy [1, 11]. Although irregular connectivity patterns have been studied before [3, 4, 21, 22, 26], to the best of our knowledge we are the first to propose register and consensus implementations in the presence of flaky channels.

The implementation for consensus is the more challenging one (§5). Since, as we explained above, failure and leader detectors are not useful in the presence of flaky channels, we take a different approach: we generalize the abstraction of a *view synchronizer* [12, 13, 40, 41], recently proposed for Byzantine consensus, to benign failure settings with flaky channels. Roughly, a view synchronizer facilitates a commonly used design pattern in which the protocol execution is divided into *views* (aka *rounds*). Each view has a designated leader that coordinates

the process interactions within that view. The task of the synchronizer is to ensure that sufficiently many correct processes are eventually able to spend sufficient time in the same view with a correct leader, which would then be able to drive consensus to completion. Supporting this functionality under partial synchrony is nontrivial, as during asynchronous periods clocks can diverge and messages used to synchronize views could get lost or delayed. View synchronizers encapsulate the necessary logic to deal with these complexities, thereby enabling modular design of consensus protocols.

In contrast to failure or leader detectors, a synchronizer does not attempt to identify the set of misbehaving processes (which, as we argued above, is impossible with flaky channels), but instead delegates this task to the top-level protocol. This can monitor the current leader using timeouts and other protocol-specific logic and ask the synchronizer to switch to another view with a different leader if it detects a lack of progress. For example, if processes 1 and 3 in Figure 1c do not observe progress on behalf of process 2 due to message loss, they would eventually initiate a view change and try another leader.

We present a specification of a view synchronizer sufficient to implement consensus in the presence of flaky channels, its implementation, and prove that the implementation satisfies our specification. Even though handling crashes is easier than Byzantine faults, flaky channels create another challenge: processes outside the connected core (such as process 2 in Figures 1b-c) falsely suspecting the current leader should not be able to force a view change, disrupting a working view. Using our synchronizer we then design a consensus protocol that tolerates flaky channels and prove its correctness.

Finally, we also demonstrate how our lower and upper bounds can be applied to establish consensus solvability in various existing models of weak connectivity [2, 3, 24, 32, 38]. In particular, we show that some connectivity models strong enough to implement the Ω leader detector are nevertheless too weak to implement consensus (§6).

2 System Model

We consider a set $\mathcal{P}$ of n processes which can fail by crashing. A process is *correct* if it never crashes, and *faulty* otherwise. Processes communicate by exchanging messages through a set of unidirectional channels $\mathcal{C}$: for every pair of processes $p, q \in \mathcal{P}$ there is a channel $(p, q) \in \mathcal{C}$ for sending messages from p to q.

We consider two notions of channel correctness (*reliable* and *eventually reliable*) and two notions of channel faultiness (*disconnected* and *flaky*). Given correct processes p and q:

- a channel (p, q) is *reliable* if it delivers every message sent by p to q;
- a channel (p, q) is *eventually reliable* if there exists a time t such that the channel delivers every message sent by p to q after t;
- a channel (p, q) is *disconnected* if it drops all messages sent by p to q; and
- a channel (p, q) is *flaky* if it drops an arbitrary subset of messages sent by p to q.

Flakiness is a very broad failure mode: it subsumes disconnections and allows the channel to choose which messages to drop, without any guarantee of fairness. Thus, flaky channels are strictly more permissive than fair lossy, which are computationally equivalent to reliable, as we explained in §1.

Our lower bounds assume the stronger notion of channel correctness (reliable) and the more restrictive notion of faultiness (disconnected), whereas our upper bounds assume weaker correctness (eventually reliable) and broader faultiness (flaky). We combine these channel reliability assumptions with two types of synchrony guarantees for correct channels, which yields four models used in our results (Table 1). We use the *asynchronous* model in our

results about registers, and the *partially synchronous* model [17, 23] in our results about consensus. The latter assumes that there exists a *global stabilization time* GST and a duration δ such that after GST message delays between correct processes connected by correct channels are bounded by δ. However, messages sent before GST can get arbitrarily delayed. Partial synchrony also assumes that processes have clocks that can drift unboundedly from the real time before GST, but do not drift thereafter. Both GST and δ are unknown to our protocols.

To state our results we need an ability to restrict which processes and channels can fail. We do this by generalizing the classical notion of a *fail-prone system* [39], which we formulate in a generic fashion irrespective of a specific channel failure type (flaky or disconnected). We specify the failure type when stating our lower or upper bounds. A *failure pattern* is a pair $(P, C) \in 2^{\mathcal{P}} \times 2^{\mathcal{C}}$ that defines which processes and channels fail in a single execution. We assume that C only contains channels between pairs of correct processes, since those incident to faulty processes fail automatically: $(p, q) \in C \implies \{p, q\} \cap P = \emptyset$. For a failure pattern $f = (P, C)$, an execution σ of the system is *f-compliant* if exactly the processes in P and the channels in C fail in σ. A *fail-prone system* $\mathcal{F}$ is a set of failure patterns. For example, a standard fail-prone system where any minority of processes can fail and channels between correct processes cannot fail corresponds to $\mathcal{F}_M = \{(Q, \emptyset) \mid Q \subseteq \mathcal{P} \wedge |Q| \leq \lfloor \frac{n-1}{2} \rfloor\}$.

3 Correctness Properties

We consider algorithms $\mathcal{A}$ that implement shared memory objects $\mathcal{O}$, such as a register or consensus, in the models just introduced. For an arbitrary domain of values *Val*, the *register* interface consists of operations *write*(v), $v \in$ *Val* and *read*(), which return *ack* and a value in *Val*, respectively. The interface of a consensus object consists of a single operation *propose*(v), $v \in$ *Val*, which returns a value in *Val*.

Safety. The consensus object is required to satisfy the standard safety properties: (*Agreement*) all terminating *propose* invocations must return the same value; and (*Validity*) *propose* can only return a value passed to some *propose* invocation.

We now specify the safety properties for registers. An operation op' *follows* an operation op, denoted $op \rightarrow op'$, if op' is invoked after op returns; op' is *concurrent* with op if neither $op \rightarrow op'$ nor $op' \rightarrow op$. The $\mathcal{O}$'s *sequential specification* specifies the behavior of $\mathcal{O}$ in the executions in which no operations are concurrent with each other. The sequential specification of a register requires every read to return the value written by the latest preceding write operation. An execution σ of $\mathcal{A}$ is *linearizable* [30] if there exists a set of responses X and a sequence $\pi = op_1, op_2, \ldots$ of all complete operations in σ and some subset of incomplete operations paired with responses in X such that π complies with $\mathcal{O}$'s sequential specification and satisfies $op_i \rightarrow op_j \implies i < j$. An execution σ of a register satisfies *safeness* if the subsequence of σ consisting of all write operations and all read operations not concurrent with any writes is linearizable. An implementation $\mathcal{A}$ of a register is *safe* [34] if each one of its executions satisfies safeness; $\mathcal{A}$ is *atomic* if each one of its executions is linearizable.

Liveness. Due to faulty channels, some correct processes may not be able to terminate, e.g., if they are partitioned off from the rest of the system. Therefore, to state liveness we parameterize the classical notions of obstruction-freedom and wait-freedom by the failures allowed and the subsets of correct processes where termination is required. We use the weaker obstruction-freedom in our lower bounds and the stronger wait-freedom in our upper bounds.

For a failure pattern $f = (P, C)$, we say that $\mathcal{A}$ is (f, T)-*wait-free* if $T \subseteq \mathcal{P} \setminus P$ and for every process $p \in T$, operation *op* and f-compliant fair execution σ of $\mathcal{A}$, if *op* is invoked by p in σ, then *op* eventually returns. An operation *op eventually executes solo* in an execution σ if either *(i) op* returns in σ, or *(ii)* there exists a suffix σ' of σ such that for all operations op', if op' is concurrent with *op* in σ, then the process that invoked op' is crashed in σ'. We say that $\mathcal{A}$ is (f, T)-*obstruction-free* if $T \subseteq \mathcal{P} \setminus P$ and for every process $p \in T$, operation *op* and f-compliant fair execution σ of $\mathcal{A}$, if *op* is invoked by p and eventually executes solo in σ, then *op* eventually returns. Note that our notion of obstruction-freedom mirrors its well-known shared memory counterparts, such as solo termination [25] and the formalization of obstruction-freedom [31] given in [9].

We next lift these notions to a fail-prone system $\mathcal{F}$ and a *termination mapping* $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ – a function mapping each failure pattern to the set of correct processes whose operations are required to terminate (thus, $\forall f = (P, C) \in \mathcal{F}.\ \tau(f) \subseteq \mathcal{P} \setminus P$). Namely, we say that $\mathcal{A}$ is $(\mathcal{F}, \tau)$-*obstruction-free* (respectively, *wait-free*) if for every $f \in \mathcal{F}$, $\mathcal{A}$ is $(f, \tau(f))$-obstruction-free (respectively, wait-free). For example, the standard guarantee of wait-freedom under a minority of processes failures corresponds to $(\mathcal{F}_M, \tau_M)$-wait-freedom, where $\mathcal{F}_M$ is defined in §2 and τ_M is such that $\forall f = (P, \emptyset) \in \mathcal{F}_M.\, \tau_M(f) = \mathcal{P} \setminus P$.

4 Inherent Connectivity Requirements for Registers and Consensus

We now investigate which connectivity requirements are necessary to implement first registers and then consensus in the models of Table 1. We start with a few simple results that serve as a stepping stone for our later lower bounds. These results are also of independent interest because they generalize the celebrated CAP theorem [14]: it is impossible to guarantee all of Consistency, Availability and network Partition-tolerance.

4.1 CAP Revisited

The CAP formalization by Gilbert and Lynch [27] considers an asynchronous system without process failures. It models consistency as implementing an atomic register, availability as providing wait-freedom at all processes, and partition-tolerance as tolerating channels that can lose any messages (in our terminology, flaky).

▶ **Theorem 1** (CAP). *It is impossible to implement a wait-free atomic register in a message-passing system where processes do not fail, but all channels are flaky.*

Despite its fame, the CAP theorem yields only a weak lower bound. First, it is not tight: the theorem only says that, to implement a wait-free atomic register, *some* channels must be correct, but obviously, a single correct channel would not be sufficient in general. Second, CAP requires availability at *all* processes and thus is not applicable if we are trying to ensure availability at least in *a part* of the system, as formalized by the notions of liveness in §3. Our first result lifts these limitations and also strengthens CAP in other ways: it considers a weaker object, obstruction-free safe register, and a stronger system model $\mathcal{M}_{\text{ARD}}$, where channels can only fail by disconnection. Informally, we show that, in this setting, all processes where obstruction-freedom holds must be transitively connected via correct channels: no such process can be partitioned off from the rest.

Formally, let $\mathcal{G} = (\mathcal{P}, \mathcal{C})$ be the directed graph constructed by taking processes as vertices and channels as edges. For a failure pattern $f = (P, C)$, let $\mathcal{G} \setminus f$ denote the subgraph of $\mathcal{G}$ obtained by removing all processes in P along with their incident channels, as well as all channels in C.

▶ **Theorem 2.** *Let f be a failure pattern and $T \subseteq \mathcal{P}$. If some algorithm $\mathcal{A}$ implements an (f,T)-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then T is strongly connected in $\mathcal{G} \setminus f$.*

Before proving the theorem, we note some of its consequences. The theorem yields the following corollary in the special case when the failure pattern f disallows process failures and obstruction-freedom is required at all processes.

▶ **Corollary 3.** *Let f be a failure pattern that disallows process failures: $\exists C \subseteq \mathcal{C}. f = (\emptyset, C)$. If some algorithm $\mathcal{A}$ implements an $(f,\mathcal{P})$-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then the graph $\mathcal{G} \setminus f$ is strongly connected.*

This implies CAP (Theorem 1): if an algorithm $\mathcal{A}$ implements an atomic wait-free register, then it also implements an obstruction-free safe register, and by the above corollary, all processes must be strongly connected by correct channels. Corollary 3 also yields a tight lower bound. To see this, consider its lifting to an arbitrary fail-prone system $\mathcal{F}$ and the termination mapping $\tau_{\mathcal{P}} = \lambda f. \mathcal{P}$:

▶ **Corollary 4.** *Let $\mathcal{F}$ be a fail-prone system that disallows process failures: $\forall f \in \mathcal{F}. \exists C \subseteq \mathcal{C}. f = (\emptyset, C)$. If some algorithm $\mathcal{A}$ implements an $(\mathcal{F}, \tau_{\mathcal{P}})$-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then for all $f \in \mathcal{F}$ the graph $\mathcal{G} \setminus f$ is strongly connected.*

Then the following proposition implies a matching upper bound: it shows that a wait-free atomic register can be implemented in the more adversarial model $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky). The proposition follows from a more general result we present later (Theorem 8).

▶ **Proposition 5.** *Let $\mathcal{F}$ be a fail-prone system that disallows process failures and such that for all $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ is strongly connected. Then there exists an algorithm $\mathcal{A}$ implementing an $(\mathcal{F}, \tau_{\mathcal{P}})$-wait-free atomic register over $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky).*

Proof of Theorem 2. Assume by contradiction that $\mathcal{A}$ is an (f,T)-obstruction-free implementation of a safe register, but T is not strongly connected in $\mathcal{G} \setminus f$. Then for some $u, v \in T$, either there is no path from u to v or from v to u in $\mathcal{G} \setminus f$. Without loss of generality we assume the former. Let S_u be the strongly connected component of $\mathcal{G} \setminus f$ containing u, and S_v the strongly connected component of $\mathcal{G} \setminus f$ containing v; then $S_u \cap S_v = \emptyset$. Let R_u be the set of processes outside S_u that can reach u in $\mathcal{G} \setminus f$, and R_v the set of processes outside S_v that can reach v (see Figure 2).

▷ Claim 1. For any $k \in \{u, v\}$, $R_k \cup S_k$ is unreachable from $\mathcal{P} \setminus (R_k \cup S_k)$ in $\mathcal{G} \setminus f$.

▷ Claim 2. For any $k \in \{u, v\}$, R_k is unreachable from S_k in $\mathcal{G} \setminus f$.

The above claims easily follow from the definitions of R_k and S_k.

▷ Claim 3. $S_u \cap (R_v \cup S_v) = \emptyset$.

Proof. Assume by contradiction that $S_u \cap (R_v \cup S_v) \neq \emptyset$ and let $w \in S_u \cap (R_v \cup S_v)$. Because $w \in S_u$, there exists a path from u to w. Because $w \in R_v \cup S_v$, there exists a path from w to v. By concatenating these two paths we get a path from u to v in $\mathcal{G} \setminus f$, contradicting our assumption that there is no such path. ◁

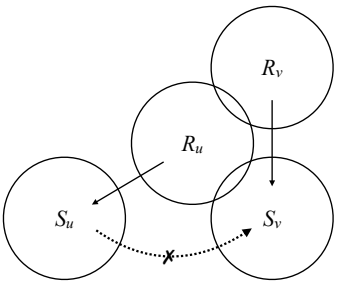


Figure 2 Illustration of the sets used in the proof of Theorem 2.

Let α_1 be a fair execution of $\mathcal{A}$ where processes and channels in f fail at the beginning, the process u invokes a *read* operation, and no other operation is invoked in α_1. Because $u \in T$ and $\mathcal{A}$ is (f,T)-obstruction-free, the *read* operation must eventually terminate. Since there are no *write* invocations, the *read* must return 0 – the initial value of the register. Let α_2 be the prefix of α_1 ending with this response. By Claim 1, $R_u \cup S_u$ is unreachable from $\mathcal{P} \setminus (R_u \cup S_u)$. Thus, the actions by processes in $R_u \cup S_u$ do not depend on those by processes in $\mathcal{P} \setminus (R_u \cup S_u)$. Then the projection of α_2 to actions by processes in $R_u \cup S_u$ is an execution of $\mathcal{A}$: $\alpha_3 = \alpha_2|_{R_u \cup S_u}$. By Claim 2, R_u is unreachable from S_u. Thus, the actions by processes in R_u do not depend on those by processes in S_u. Then $\alpha = \alpha_3|_{R_u}\alpha_3|_{S_u}$ is also an execution of $\mathcal{A}$. Notice that $\alpha_3|_{R_u}$ does not contain any operation invocation, but it contains steps taken by processes upon startup.

Let β_1 be a fair execution of $\mathcal{A}$ where processes and channels in f fail at the beginning. The execution β_1 starts with all the actions from $\alpha_3|_{R_u}$ followed by a *write*(1) invocation by process v, and no other operation is invoked in β_1. Because $v \in T$ and $\mathcal{A}$ is (f,T)-obstruction-free, the *write* operation must eventually terminate. Let β_2 be the prefix of β_1 ending with this response. By Claim 1, $R_u \cup S_u$ is unreachable from $\mathcal{P} \setminus (R_u \cup S_u)$. By Claim 2, R_u is unreachable from S_u. Therefore, R_u is unreachable from $\mathcal{P} \setminus R_u$. By Claim 1, $R_v \cup S_v$ is unreachable from $\mathcal{P} \setminus (R_v \cup S_v)$. Therefore, $R_u \cup R_v \cup S_v$ is unreachable from $\mathcal{P} \setminus (R_u \cup R_v \cup S_v)$. Thus, the actions by processes in $R_u \cup R_v \cup S_v$ do not depend on those by processes in $\mathcal{P} \setminus (R_u \cup R_v \cup S_v)$. Then $\beta = \beta_2|_{R_u \cup R_v \cup S_v}$ is an execution of $\mathcal{A}$. Recall that β_1 starts with $\alpha_3|_{R_u}$, and hence, so does β. Let δ be the suffix of β such that $\beta = \alpha_3|_{R_u}\delta$.

Consider the execution $\sigma = \alpha_3|_{R_u}\delta\alpha_3|_{S_u}$ where the actions occur before processes and channels in f fail. By Claim 3, $S_u \cap (R_v \cup S_v) = \emptyset$, and by the definition of R_u, we have $S_u \cap R_u = \emptyset$. Hence, $S_u \cap (R_u \cup R_v \cup S_v) = \emptyset$. Then, given that δ only contains actions by processes in $R_u \cup R_v \cup S_v$, we get: $\sigma|_{R_u \cup R_v \cup S_v} = (\alpha_3|_{R_u}\delta\alpha_3|_{S_u})|_{R_u \cup R_v \cup S_v} = \alpha_3|_{R_u}\delta = \beta$ and $\sigma|_{S_u} = (\alpha_3|_{R_u}\delta\alpha_3|_{S_u})|_{S_u} = \alpha_3|_{S_u} = (\alpha_3|_{R_u}\alpha_3|_{S_u})|_{S_u} = \alpha|_{S_u}$. Therefore, σ is indistinguishable from β to the processes in $R_u \cup R_v \cup S_v$ and from α to the processes in S_u. Finally, $\sigma|_{\mathcal{P} \setminus (R_u \cup S_u \cup R_v \cup S_v)} = \varepsilon$.

Thus, for every process, σ is indistinguishable to this process from some execution of $\mathcal{A}$. Furthermore, each message received by a process in σ has previously been sent by another process. Therefore, σ is an execution of $\mathcal{A}$. However, in this execution *write*(1) terminates before a *read* that fetches 0 is invoked. This contradicts the assumption that $\mathcal{A}$ implements a safe register. The contradiction shows that T must be strongly connected in $\mathcal{G} \setminus f$. ◀

4.2 Connectivity Requirements under Bounded Process Failures

Consider a straightforward lifting of Theorem 2 to fail-prone systems:

▶ **Corollary 6.** *Let $\mathcal{F}$ be a fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ a termination mapping. If some algorithm $\mathcal{A}$ implements an $(\mathcal{F}, \tau)$-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then for all $f \in \mathcal{F}$, $\tau(f)$ is strongly connected in $\mathcal{G} \setminus f$.*

This result does not make any assumptions about the fail-prone system, and its CAP-like specialization given by Corollary 4 only considers fail-prone systems without process failures. However, algorithms are commonly designed to tolerate a bounded number of process failures, and we next show that stronger connectivity requirements are necessary in this case. To formalize this class of algorithms, we use the following notion. A *k-fail-prone system* $\mathcal{F}$ allows any set of k processes or fewer to fail, but disallows failures of more than k processes:

$$(\forall P \subseteq \mathcal{P}.\ |P| \leq k \implies \exists C \subseteq \mathcal{C}.\ (P, C) \in \mathcal{F}) \wedge (\forall (P, C) \in \mathcal{F}.\ |P| \leq k).$$

For example, the system $\mathcal{F}_M$ from §2 is $\lfloor \frac{n-1}{2} \rfloor$-fail-prone.

The following theorem establishes minimal connectivity constraints required to implement a safe register under the model $\mathcal{M}_{\text{ARD}}$ in the presence of k process crashes. It assumes a particularly weak termination guarantee which only requires obstruction-freedom to hold at some *non-empty* set of processes for each failure pattern. The theorem states that, no matter how small this set is, it must be part of a set of $> k$ correct processes strongly connected by correct channels.

▶ **Theorem 7.** *Let $\mathcal{F}$ be a k-fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ a termination mapping such that $\forall f = (P, C) \in \mathcal{F}. \tau(f) \neq \emptyset$. Assume that some algorithm $\mathcal{A}$ implements an $(\mathcal{F}, \tau)$-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected). Then for all $f \in \mathcal{F}$, there exists a strongly connected component of $\mathcal{G} \setminus f$ that contains $\tau(f)$ and has a cardinality greater than k.*

Proof. Let $f \in \mathcal{F}$ and S be the strongly connected component of $\mathcal{G} \setminus f$ containing $\tau(f)$. Assume by contradiction that $|S| \leq k$. Pick an arbitrary process $p \in \tau(f)$; then p is correct according to f. Let α_1 be a fair execution of $\mathcal{A}$ where processes and channels in f fail at the beginning, the process p invokes a *read* operation, and no other operation is invoked in α_1. Because $p \in \tau(f)$ and $\mathcal{A}$ is $(\mathcal{F}, \tau)$-obstruction-free, the *read* operation must eventually terminate. Since there are no *write* invocations, the *read* must return 0 – the initial value of the register. Let α_2 be the prefix of α_1 ending with this response. Let R be the set of processes outside S that can reach S in $\mathcal{G} \setminus f$. Similarly to the proof of Theorem 2, the definitions of R and S imply the following claims.

▷ Claim 1. $R \cup S$ is unreachable from $\mathcal{P} \setminus (R \cup S)$ in $\mathcal{G} \setminus f$.

▷ Claim 2. R is unreachable from S in $\mathcal{G} \setminus f$.

Claim 1 implies that the actions by processes in $R \cup S$ do not depend on those by processes in $\mathcal{P} \setminus (R \cup S)$. Then $\alpha_3 = \alpha_2|_{R \cup S}$ is an execution of $\mathcal{A}$. Claim 2 implies that the actions by processes in R do not depend on those by processes in S. Then $\alpha = \alpha_3|_R \alpha_3|_S$ is also an execution of $\mathcal{A}$.

Since $|S| \leq k$ and $\mathcal{F}$ is a k-fail-prone system, there exists $C' \subseteq \mathcal{C}$ such that $f' = (S, C') \in \mathcal{F}$. Pick an arbitrary process $q \in \tau(f')$; then $q \notin S$. Let β_1 be a fair execution of $\mathcal{A}$ where processes and channels in f' fail at the beginning. The execution β_1 starts with all the actions from $\alpha_3|_R$ followed by a *write*(1) invocation by process q, and no other operation is invoked in β_1. Because $q \in \tau(f')$ and $\mathcal{A}$ is $(\mathcal{F}, \tau)$-obstruction-free, the *write* operation must eventually terminate. Let β be the prefix of β_1 ending with this response and let δ be the suffix of β such that $\beta = \alpha_3|_R \delta$.

Consider the execution $\sigma = \alpha_3|_R \delta \alpha_3|_S$ where the actions occur before processes and channels in f fail. Given that δ does not contain any actions by processes in S, we get: $\sigma|_S = \alpha_3|_S = (\alpha_3|_R \alpha_3|_S)|_S = \alpha|_S$ and $\sigma|_{\mathcal{P}\setminus S} = \alpha_3|_R \delta = \beta$. Therefore, σ is indistinguishable from α to the processes in S and from β to the processes in $\mathcal{P} \setminus S$.

Thus, for every process, σ is indistinguishable to this process from some execution of $\mathcal{A}$. Furthermore, each message received by a process in σ has previously been sent by another process. Therefore, σ is an execution of $\mathcal{A}$. However, in this execution *write*(1) terminates before a *read* that fetches 0 is invoked. This contradicts the assumption that $\mathcal{A}$ implements a safe register. The contradiction derives from assuming that $|S| \leq k$, so that $|S| > k$. ◀

Theorem 7 is most interesting in the common practical case of $n = 2k + 1$, which is the minimal number of processes needed to tolerate k crashes in asynchronous registers [37] and partially synchronous consensus [23]. In this case the theorem ensures that for each failure pattern f, the graph $\mathcal{G} \setminus f$ has a strongly connected component containing $\geq k + 1$ processes. More generally, for arbitrary $\mathcal{G}$ and f, we call a strongly connected component of $\mathcal{G} \setminus f$ containing a majority of processes in $\mathcal{G}$ a *connected core* of the graph. It is easy to see there can exist at most one connected core for given $\mathcal{G}$ and f. For example, in Figure 1a the connected core is $\{1, 2, 3\}$, whereas in Figures 1b-c it is $\{1, 3\}$. As we now show, the lower bound of Theorem 7 is tight for $n = 2k + 1$. In fact, assuming the existence of a connected core, we can implement an atomic register that is wait-free at all members of the connected core under the more adversarial model $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky).

▶ **Theorem 8.** *Let $\mathcal{F}$ be a fail-prone system such that for all $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ contains a connected core $\mathcal{S}_f$, and let $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be the termination mapping such that $\forall f \in \mathcal{F}$. $\tau(f) = \mathcal{S}_f$. Then there exists an $(\mathcal{F}, \tau)$-wait-free implementation of an atomic register over the model $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky).*

We defer the proof of the theorem to [42, §A]. The proof constructs the desired implementation as a variant of ABD [8] that uses gossip-style data propagation to deal with indirect connectivity. The most interesting aspect of this implementation is that, despite the need for gossip, it uses only bounded space. We illustrate the technique for bounding space when presenting our consensus implementation in §5.

4.3 Connectivity Requirements for Consensus

The lower bounds in the previous section also apply to consensus under partial synchrony, with analogs of Theorems 2 and 7 formulated as follows:

▶ **Theorem 9.** *Let f be a failure pattern and $T \subseteq \mathcal{P}$. If some algorithm $\mathcal{A}$ is an (f, T)-obstruction-free implementation of consensus over $\mathcal{M}_{\text{PRD}}$ (partially synchronous / reliable / disconnected), then T is strongly connected in $\mathcal{G} \setminus f$.*

▶ **Theorem 10.** *Let $\mathcal{F}$ be a k-fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be a termination mapping such that $\forall f = (P, C) \in \mathcal{F}$. $\tau(f) \neq \emptyset$. Assume that some algorithm $\mathcal{A}$ is an $(\mathcal{F}, \tau)$-obstruction-free implementation of consensus over $\mathcal{M}_{\text{PRD}}$ (partially synchronous / reliable / disconnected). Then for all $f \in \mathcal{F}$, there exists a strongly connected component of $\mathcal{G} \setminus f$ that contains $\tau(f)$ and has a cardinality greater than k.*

To see why these theorems hold observe first that Theorems 2 and 7 also hold if the model $\mathcal{M}_{\text{ARD}}$ is replaced with $\mathcal{M}_{\text{PRD}}$: since the executions σ constructed in the proofs of the theorems are finite, they are also valid executions under $\mathcal{M}_{\text{PRD}}$ where all actions occur before GST. Then the required follows from the fact that registers can be implemented from consensus. We formally prove this for our setting in [42, §B].

Finally, similarly to Theorem 8, for the case of $n = 2k + 1$ we can prove an upper bound matching Theorem 10 in the model $\mathcal{M}_{\text{PEF}}$ (partially synchronous / eventually reliable / flaky). This result is much more difficult than the upper bound for registers, and we prove it in the next section.

▶ **Theorem 11.** *Let $\mathcal{F}$ be a fail-prone system such that for all $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ contains a connected core $\mathcal{S}_f$, and let $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be the termination mapping such that $\forall f \in \mathcal{F}.\ \tau(f) = \mathcal{S}_f$. Then there exists an $(\mathcal{F}, \tau)$-wait-free implementation of consensus over the model $\mathcal{M}_{\text{PEF}}$ (partially synchronous / eventually reliable / flaky).*

5 Consensus in the Presence of Flaky Channels

We now present a consensus protocol in the model $\mathcal{M}_{\text{PEF}}$ that validates Theorem 11. Consider a fail-prone system $\mathcal{F}$ satisfying the conditions of the theorem: for each $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ contains a connected core. For the remainder of the section we fix a failure pattern $f \in \mathcal{F}$ and let $\mathcal{S}$ be the corresponding connected core. Since, as we explained in §1, classical failure and leader detectors are not useful in the presence of flaky channels, we take a different approach. Our protocol is implemented on top of a *view synchronizer* [12, 13, 40, 41], which enables the processes to divide their execution into a series of views, each with a designated leader. At a high level, the synchronizer's goal is to bring sufficiently many correct processes with enough connectivity (e.g., those from $\mathcal{S}$) into a view led by a well-connected leader and keep them in that view for sufficiently long to reach an agreement. Supporting this in the presence of flaky channels is nontrivial as the processes outside the connected core (such as process 2 in Figures 1b-c) may fail to observe progress on behalf of the leader of a functional view and request a premature view change. We first present the specification (§5.1) and the implementation (§5.2) of a synchronizer that addresses this challenge. We then use it to construct a consensus protocol (§5.3) satisfying the requirements of Theorem 11.

5.1 Synchronizer Specification

We consider a synchronizer interface defined in [12, 40]. Let $\mathsf{View} = \{1, 2, \dots\}$ be the set of *views*, ranged over by v; we use 0 to denote an invalid initial view. The synchronizer produces notifications $\mathsf{new\_view}(v)$ at a process, telling it to enter a view v. To trigger these, the synchronizer allows a process to call a function $\mathsf{advance}()$, which signals that the process wishes to *advance* to a higher view. We assume that a process does not call $\mathsf{advance}()$ twice without an intervening $\mathsf{new\_view}$ notification.

In Figure 3 we give a specification of a view synchronizer for the system model $\mathcal{M}_{\text{PEF}}$, which is an adaptation of the one for the Byzantine setting [12]. The Monotonicity property ensures that, at any given process, its view can only increase. The Validity property ensures that a process may only enter a view $v + 1$ if some process in the connected core $\mathcal{S}$ has called $\mathsf{advance}$ in v. This prevents processes with bad connectivity from disrupting $\mathcal{S}$ by forcing view changes (e.g., process 2 in Figure 1c, where $\mathcal{S} = \{1, 3\}$). The Bounded Entry property ensures that, if some process from $\mathcal{S}$ enters a view v, then all processes from $\mathcal{S}$ will do so at most d units of time of each other (for some constant d). This only holds if within d no process from $\mathcal{S}$ attempts to advance to a higher view, as this may make some processes from $\mathcal{S}$ skip v and enter a higher view directly. Bounded Entry only holds starting from some view $\mathcal{V}$, since a synchronizer may not be able to guarantee it for views entered before GST. The Startup property ensures that if more than $\frac{n}{2}$ processes from $\mathcal{S}$ attempt to advance from view 0, then some process from $\mathcal{S}$ will enter view 1. The Progress property determines the

- **Monotonicity.** A process may only enter increasing views:
  $\forall i, v, v'.\ E_i(v)\downarrow\ \wedge\ E_i(v')\downarrow\ \implies\ (v < v'\ \iff\ E_i(v) < E_i(v'))$
- **Validity.** A process only enters $v+1$ if some process from $\mathcal{S}$ has attempted to advance from v:
  $\forall i, v.\ E_i(v+1)\downarrow\ \implies\ A^{\mathcal{S}}_{\text{first}}(v)\downarrow\ \wedge\ A^{\mathcal{S}}_{\text{first}}(v) < E_i(v+1)$
- **Bounded Entry.** For some $\mathcal{V}$ and d, if a process from $\mathcal{S}$ enters $v \geq \mathcal{V}$ and no process from $\mathcal{S}$ attempts to advance to a higher view within d, then every process from $\mathcal{S}$ will enter v within d:
  $\exists \mathcal{V}, d.\ \forall v \geq \mathcal{V}.\ E^{\mathcal{S}}_{\text{first}}(v)\downarrow\ \wedge\ \neg(A^{\mathcal{S}}_{\text{first}}(v) < E^{\mathcal{S}}_{\text{first}}(v) + d)\ \implies$
  $(\forall p_i \in \mathcal{S}.\ E_i(v)\downarrow)\ \wedge\ (E^{\mathcal{S}}_{\text{last}}(v) \leq E^{\mathcal{S}}_{\text{first}}(v) + d)$
- **Startup.** If $> \frac{n}{2}$ processes from $\mathcal{S}$ invoke advance, then some process from $\mathcal{S}$ will enter view 1:
  $(\exists P \subseteq \mathcal{S}.\ |P| > \frac{n}{2}\ \wedge\ (\forall p_i \in P.\ A_i(0)\downarrow))\ \implies\ E^{\mathcal{S}}_{\text{first}}(1)\downarrow$
- **Progress.** If a process from $\mathcal{S}$ enters v and, for some set $P \subseteq \mathcal{S}$ of $> \frac{n}{2}$ processes, any process in P that enters v eventually invokes advance, then some process from $\mathcal{S}$ will enter $v+1$:
  $\forall v.\ E^{\mathcal{S}}_{\text{first}}(v)\downarrow\ \wedge\ (\exists P \subseteq \mathcal{S}.\ |P| > \frac{n}{2}\ \wedge\ (\forall p_i \in P.\ E_i(v)\downarrow\ \implies\ A_i(v)\downarrow))\ \implies\ E^{\mathcal{S}}_{\text{first}}(v+1)\downarrow$

Notation:

- $E_i(v)$: the time when process p_i enters a view v
- $E^{\mathcal{S}}_{\text{first}}(v)$, $E^{\mathcal{S}}_{\text{last}}(v)$: the earliest and the latest time when a process from $\mathcal{S}$ enters a view v
- $A_i(v)$, $A^{\mathcal{S}}_{\text{first}}(v)$, $A^{\mathcal{S}}_{\text{last}}(v)$: similarly for times of attempts to advance from a view v
- $g(x)\downarrow$, $g(x)\uparrow$: $g(x)$ is defined/undefined

Figure 3 Synchronizer properties satisfied in executions with connected core $\mathcal{S}$.

conditions under which some process from $\mathcal{S}$ will enter a view $v+1$: this will happen if a process from $\mathcal{S}$ enters the view v, and for some set P of more than $\frac{n}{2}$ processes from $\mathcal{S}$, any process in P entering v eventually invokes advance (e.g., 1 and 3 in Figure 1c).

As we show below, the above properties work in tandem to ensure the liveness of consensus in the presence of flaky channels. Informally, Progress allows processes to iterate over views in search for one with a well-connected leader; Bounded Entry enables all processes in the connected core to promptly enter this view; and Validity ensures that these processes can stay in it despite any disruption from processes with flaky connectivity.

5.2 Synchronizer Implementation

In Figure 4 we present an algorithm that implements the specification in Figure 3 in the model $\mathcal{M}_{\text{PEF}}$. This implementation requires only bounded space, despite the fact that correct channels in $\mathcal{M}_{\text{PEF}}$ are only *eventually* reliable, and thus can lose messages before GST. A process stores its current view in a variable curr_view. A process also maintains an array views tracking, for every other process, the highest view to which it wishes to advance. When the process invokes advance (line 1), the synchronizer does not immediately switch to the next view. Instead, it updates its entry in the views array and propagates the whole array in a `WISH` message, advertising its wish to advance (line 3). Upon the receipt of a `WISH` message (line 6), a process incorporates the information received into its views array, keeping entries with the highest view (line 7). This mechanism ensures that information is propagated between processes that do not have direct connectivity via a correct channel.

Since the membership of $\mathcal{S}$ is unknown to the processes, to satisfy Validity the synchronizer cannot initiate a view change based on an advance() call by a single process: the synchronizer cannot tell whether this process is from $\mathcal{S}$ or not. Instead, we require wishes to advance from

```
1  function advance():
2      views[i] ← curr_view + 1
3      send WISH(views) to all

4  periodically                ▷ every ρ time units
5      send WISH(views) to all

6  when received WISH(V)
7      for p_j ∈ P do views[j] ← max(views[j], V[j])
8      v' ← max{v | ∃p_j. views[j] = v ∧ |{p_k | views[k] ≥ v}| > n/2}
9      if v' > curr_view then
10         curr_view ← v'
11         trigger new_view(v')
12         send WISH(views) to all
```

Figure 4 Synchronizer at a process p_i.

a majority of processes, so that at least one of them must be from $\mathcal{S}$. In more detail, upon receiving a WISH message, we compute v' as the $(\lfloor \frac{n}{2} \rfloor + 1)$-st highest view in views (line 8). Thus, at least one process from $\mathcal{S}$ wishes to advance to a view $\geq v'$. In this case the current process enters v' if this view is greater than its curr_view (line 11). Note that a process may be forced to switch views even if it did not invoke advance; this helps lagging processes to catch up. To satisfy Bounded Entry, the process disseminates the information that made it enter the new view (line 12) to ensure that other processes also do so promptly. Finally, to deal with message loss before GST, a process periodically resends its views array (line 4).

We can also prove that the synchronizer satisfies Progress: this property requires $> \frac{n}{2}$ advance calls by processes in $\mathcal{S}$, which are well-connected enough for the corresponding WISHes to eventually propagate within $\mathcal{S}$ and enable the guard at line 9 (e.g., processes 1 and 3 in Figure 1c). Note that Progress wouldn't hold if it required $> \frac{n}{2}$ advance calls by any processes, not necessarily in $\mathcal{S}$ (e.g., processes 1 and 2): in this case we wouldn't be able to guarantee that all the corresponding WISHes eventually propagate. We defer the proof of correctness of the synchronizer to [42, §C].

▶ **Theorem 12.** *Let $\mathcal{F}$ be a fail-prone system such that for each $f \in \mathcal{F}$, $\mathcal{G} \setminus f$ contains a connected core. Then for any $f \in \mathcal{F}$ with an associated connected core $\mathcal{S}$, every f-compliant fair execution of the algorithm in Figure 4 over the model $\mathcal{M}_{\text{PEF}}$ satisfies the properties in Figure 3.*

5.3 Consensus Protocol

In Figure 5 we present a consensus protocol in the model $\mathcal{M}_{\text{PEF}}$ (partially synchronous / eventually reliable / flaky) that validates Theorem 11. The protocol is a variation of single-decree Paxos [35] where liveness is ensured with the help of a view synchronizer. Thus, the protocol works in a succession of views produced by the synchronizer. Each view v has a fixed leader $\mathsf{leader}(v) = p_{((v-1) \mod n)+1}$ responsible for proposing a value to the other processes, which vote on the proposal. Processes monitor the leader's behavior and ask the synchronizer to advance to another view if they suspect that the leader is faulty or has a bad connectivity.

State and communication. A process stores its current view in a variable view. A variable phase tracks the progress of the process through different phases of the protocol. The initial proposal is stored in my_proposal (line 4). The process also maintains the last proposal it accepted from a leader in val, and the view in which this happened in cview.

```
1  on startup
2    advance()

3  function propose(x):
4    my_proposal ← x
5    wait until phase = DECIDED
6    return val

7  on new_view(v)
8    view ← v
9    start_timer(decision_timer, timeout)
10   M1B[i] ← (view, cview, val)
11   phase ← ENTERED

12 when the timer decision_timer expires
13   timeout ← timeout + γ
14   advance()

15 periodically                ▷ every ρ time units
16   send STATE(M1B, M2A, M2B) to all

17 when received STATE(V1B, V2A, V2B)
18   for p_j ∈ P do
19     if V1B[j].view > M1B[j].view then
20       M1B[j] ← V1B[j]
21     if V2A[j].view > M2A[j].view then
22       M2A[j] ← V2A[j]
23     if V2B[j].view > M2B[j].view then
24       M2B[j] ← V2B[j]

25 when phase = ENTERED ∧ leader(view) = p_i ∧
   |{p_j | p_j ∈ P ∧ M1B[j].view = view}| > n/2
26   Q ← {p_j | p_j ∈ P ∧ M1B[j].view = view}
27   if ∀p_j. p_j ∈ Q ⟹ M1B[j].val = ⊥ then
28     if my_proposal = ⊥ then return
29     M2A[i] ← (view, my_proposal)
30   else
31     let p_j ∈ Q be such that
         M1B[j].val ≠ ⊥ ∧
         ∀p_k ∈ Q. M1B[k].cview ≤ M1B[j].cview
32     M2A[i] ← (view, M1B[j].val)
33   phase ← PROPOSED

34 when phase ∈ {ENTERED, PROPOSED} ∧
   p_l = leader(view) ∧ M2A[l].view = view
35   (cview, val) ← M2A[l]
36   M2B[i] ← (cview, val)
37   phase ← ACCEPTED

38 when ∃v, x. v ≥ view ∧
   |{p_j | p_j ∈ P ∧ M2B[j] = (v, x)}| > n/2
39   val ← x
40   stop_timer(decision_timer)
41   phase ← DECIDED
```

Figure 5 Consensus protocol at a process p_i.

Processes exchange messages analogous to the 1B, 2A and 2B messages from Paxos, each tagged with the view where the message was issued. There is no analog of 1A messages, because leader election is controlled by the synchronizer. Since correct processes may not be directly connected by correct channels, each process has to forward the information it receives from others. Since correct channels in $\mathcal{M}_{\text{PEF}}$ are only *eventually* reliable, this furthermore has to be repeated periodically. If implemented naively, this would require unbounded space to store all the messages that need to be forwarded. Instead, we observe that it is sufficient to store, for each message type and sender, only the message of this type received from this sender with the highest view. These are stored in the arrays M1B, M2A and M2B.

For simplicity, the pseudocode in Figure 5 separates computation from communication. Most of the handlers do not send messages, but instead just modify the arrays M1B, M2A and M2B. Then instead of sending individual 1B, 2A or 2B messages like in Paxos, a process periodically sends whole arrays M1B, M2A and M2B in one big STATE message (line 16). Upon receipt of a STATE message (line 17), a process incorporates the information received into its arrays M1B, M2A and M2B, keeping entries with the highest view. This mechanism ensures that information is propagated between processes that do not have direct connectivity while using only bounded space.

Normal protocol operation. When the synchronizer tells a process to enter a new view v (line 7), the process sets view $= v$ and writes the information about the last value it accepted into its entry in the M1B array. This information will be propagated to the leader of the

view as described above. A leader waits until its M1B array contains a majority of entries corresponding to its view (line 25). Based on these, the leader computes its proposal and stores it into its entry of the M2A array. The computation is done similarly to Paxos. If some process has previously accepted a value, the leader picks the value accepted in the maximal view (line 31). Otherwise, the leader is free to propose its own value. If propose() has already been invoked at the leader, it selects my_proposal (line 29). If this has not happened yet, the leader skips its turn (line 28).

Each process waits until its M2A array contains a proposal by the leader of its view (line 34). The process then accepts the proposal by updating its val and cview. It also notifies all other processes about this by storing the information about the accepted value into its entry of the M2B array. Finally, once a process has a majority of matching entries in its M2B array (line 38), it knows that the decision has been reached, and it sets phase = DECIDED. If there is an ongoing propose() invocation, the condition at line 5 is then satisfied and the process returns the decision to the client.

Triggering view changes. We now describe when a process invokes advance(), which is key to ensuring liveness. This occurs either on startup (line 2) or when the process suspects that the current leader is faulty or has bad connectivity. To this end, when a process enters a view, it sets a decision_timer for a duration timeout (line 9) and stops the timer when a decision is reached (line 40). If the timer expires before this, the process invokes advance (line 14). A process may wrongly suspect a good leader if the timeout is initially set too low with respect to the message delay δ, unknown to the process. To deal with this, a process increases timeout whenever the timer expires (line 13).

Correctness. It remains to prove that the algorithm in Figure 5 validates Theorem 11. The proof of the safety properties of consensus is virtually identical to that of Paxos [35]. Hence, we focus on proving liveness. Here we present a proof sketch that highlights the use of the synchronizer specification and defer the proofs of auxiliary lemmas to [42, §D].

Fix a failure pattern f and let $\mathcal{S}$ be the corresponding connected core guaranteed to exist by the assumptions of Theorem 11. We prove liveness by showing that the protocol establishes properties reminiscent of those of failure detectors [17]. First, similarly to their *completeness* property, we prove that every correct process eventually attempts to advance from a view where no progress is possible (e.g., because the leader is faulty or has insufficient connectivity). We say that a process p_i *decides* in a view v if it executes line 41 while having view $= v$.

▶ **Lemma 13.** *If a correct process p_i enters a view v, never decides in v and never enters a view higher than v, then p_i eventually invokes* advance *in v.*

Informally, the lemma holds because each process monitors the progress of a view using decision_timer.

Our next lemma is similar to the *eventual accuracy* property of failure detectors. It shows that if the timeout values are high enough, then eventually any process in $\mathcal{S}$ that enters a view where progress is possible (the leader is correct and has sufficient connectivity) will not attempt to advance from it. Formally, let diameter($\mathcal{S}$) be the longest distance in the graph $\mathcal{G} \setminus f$ between two vertices in $\mathcal{S}$. Also, let $\mathcal{V}$ and d be the view and the time duration for which Bounded Entry holds (Figure 3), and let $\mathsf{timeout}_i(v)$ be the value of timeout at the process p_i while in view v.

▶ **Lemma 14.** *Let $v \geq \mathcal{V}$ be a view such that* leader$(v) \in \mathcal{S}$, $E^{\mathcal{S}}_{\text{first}}(v) \geq$ GST *and* leader(v) *invokes* propose *no later than* $E^{\mathcal{S}}_{\text{first}}(v)$. *If at each process $p_i \in \mathcal{S}$ that enters v we have* timeout$_i(v) > d + 3(\delta + \rho)$diameter$(\mathcal{S})$, *then no process in $\mathcal{S}$ invokes* advance *in v.*

Intuitively, the lemma holds because Bounded Entry ensures that all processes from $\mathcal{S}$ will enter v promptly; then since the timeouts are high enough, processes will have sufficient time to exchange the messages needed to reach a decision and stop the timers.

Proof sketch for Theorem 11. By contradiction, assume there exists $f \in \mathcal{F}$, $p_j \in \tau(f) = \mathcal{S}$ and an f-compliant fair execution of the algorithm in Figure 5 such that p_j invokes propose at a time t, but the operation never returns. Using Progress and Lemma 13, we first prove that in this case the protocol keeps moving through views forever:

▷ Claim 1. Every view is entered by some process in $\mathcal{S}$.

We next prove the following:

▷ Claim 2. Every process in $\mathcal{S}$ executes the timer expiration handler at line 12 infinitely often.

Since a process increases timeout every time decision_timer expires, by Claim 2 all processes will eventually have timeout $> d + 3(\delta + \rho)$diameter$(\mathcal{S})$. Since leaders rotate round-robin, by Claim 1 there will be infinitely many views led by p_j. Hence, there exists a view $v_1 \geq \mathcal{V}$ led by p_j such that $E^{\mathcal{S}}_{\text{first}}(v_1) \geq \max\{\mathsf{GST}, t\}$ and for any process $p_i \in \mathcal{S}$ that enters v_1 we have timeout$_i(v_1) > d + 3(\delta + \rho)$diameter$(\mathcal{S})$. Then by Lemma 14, no process in $\mathcal{S}$ calls advance in v_1. On the other hand, by Claim 1 some process in $\mathcal{S}$ enters $v_1 + 1$. Then by Validity, some process in $\mathcal{S}$ calls advance in v_1, which is a contradiction. ◀

6 Possibility and Impossibility of Classical Consensus via Ω

We now present some interesting consequences of our results. It is well-known that the failure detector Ω [16, 17] is sufficient for implementing wait-free consensus resilient to $\lfloor \frac{n-1}{2} \rfloor$ failures in non-partitionable systems under crash [17, 35], crash/recovery [5], or general omission [35, 48] failures. We now investigate if this is still the case under flaky channels and limited connectivity. Below we demonstrate that the answer depends on the amount of connectivity in the underlying network, as established by our lower and upper bounds.

Lower bound. We first show that, on the negative side, there are some weakly synchronous environments where Ω is implementable, but obstruction-free consensus is impossible even if at most one process can crash. One such environment is the system S of Aguilera et al. [3], where in every execution, all channels can be flaky except those emanating from an a priori unknown correct process (*timely source*); the latter channels are required to be eventually reliable and timely. In our framework, S is represented by the set of executions in $\mathcal{M}_{\text{PEF}}$ compliant with the following fail-prone system:

$$\mathcal{F}_S = \{(P, C) \mid |P| < n \wedge \exists p \in \mathcal{P} \setminus P.\ \forall q \in \mathcal{P} \setminus (P \cup \{p\}).\ (p, q) \notin C\}.$$

We now use Theorem 10 to prove that consensus is impossible even in a stronger variant of S where only up to a threshold k of processes are allowed to fail. Formally, for n and k such that $0 < k < n$, let $\mathcal{F}_{S,k} = \mathcal{F}_S \cap \{(P, C) \mid |P| \leq k\}$. Then $\mathcal{F}_{S,k}$ is a k-fail-prone system.

▶ **Theorem 15.** *Let n and k be such that $0 < k < n$, and $\tau : \mathcal{F}_{S,k} \to 2^{\mathcal{P}}$ be a termination mapping such that $\forall f \in \mathcal{F}_{S,k}.\ \tau(f) \neq \emptyset$. Then no algorithm can implement an $(\mathcal{F}_{S,k}, \tau)$-obstruction-free consensus in $\mathcal{M}_{\text{PEF}}$.*

Proof. By contradiction, assume such an algorithm exists. First, a usual partitioning argument similar to that of Dwork et al. [23] can be used to show that we must have $n > 2$ and $k \leq \lfloor \frac{n-1}{2} \rfloor$. Then, since $\mathcal{M}_{\text{PRD}}$ is stronger than $\mathcal{M}_{\text{PEF}}$, Theorem 10 requires that for all $f \in \mathcal{F}_{S,k}$, $\mathcal{G} \setminus f$ must contain a strongly connected component of size $> \lfloor \frac{n-1}{2} \rfloor \geq 1$. However, for a failure pattern $f = (P, C) \in \mathcal{F}_{S,k}$ such that

$$|P| = \emptyset \wedge \exists p \in \mathcal{P}.\ \forall q \in \mathcal{P} \setminus (P \cup \{p\}).\ \forall r \in \mathcal{P} \setminus P.\ (q, r) \in C,$$

all strongly connected components of $\mathcal{G} \setminus f$ are of size 1: a contradiction. ◀

Upper bounds. On the positive side, we now show that strengthening connectivity assumptions does enable consensus to be solved in many previously proposed models of weak synchrony, such as systems S^+ and S^{++} of Aguilera et al. [3] and those defined in [2,24,32,38]. These papers show that Ω can be implemented in their respective models, but (with an exception of [2]) do not give a consensus algorithm.

To show that consensus is indeed possible in these models, we introduce an intermediate model $\mathcal{S}^\star$ in which all channels can be asynchronous and flaky apart from those connecting an a priori unknown correct process (a *hub* [3]) to all the other processes in *both* directions; the latter channels are required to be eventually reliable (but can still be asynchronous). In our framework, $\mathcal{S}^\star$ is represented by the set of executions of $\mathcal{M}_{\text{AEF}}$ compliant with the following fail-prone system:

$$\mathcal{F}_{\mathcal{S}^\star} = \{(P, C) \mid |P| < n \wedge \exists p \in \mathcal{P} \setminus P.\ \forall q \in \mathcal{P} \setminus (P \cup \{p\}).\ (p, q) \notin C \wedge (q, p) \notin C\}.$$

Thus, for all $f \in \mathcal{F}_{\mathcal{S}^\star}$, $\mathcal{G} \setminus f$ is strongly connected, and in the case when at most $\lfloor \frac{n-1}{2} \rfloor$ processes fail in f, $\mathcal{G} \setminus f$ is a connected core. Then from Theorem 8, we get

▶ **Corollary 16.** *It is possible to implement a wait-free atomic register in $\mathcal{S}^\star$ if at most $\lfloor \frac{n-1}{2} \rfloor$ processes can crash.*

Since wait-free consensus tolerating any number of $< n$ process crashes can be implemented using atomic registers and Ω [36], Corollary 16 implies

▶ **Corollary 17.** *It is possible to implement a wait-free consensus in $\mathcal{S}^\star$ augmented with Ω if at most $\lfloor \frac{n-1}{2} \rfloor$ processes can crash.*

We now use this result to prove

▶ **Corollary 18.** *It is possible to implement wait-free consensus over the systems S^+ and S^{++} of [3] and the models of [2, 24, 32, 38], provided at most $\lfloor \frac{n-1}{2} \rfloor$ processes can crash.*

Proof. We first show that the systems enumerated in the theorem's statement are as strong as $\mathcal{S}^\star$. System S^+ [3] stipulates that every execution has a *fair hub*, i.e., a correct process which is connected to all other processes via fair-lossy channels [1] in both directions. Since eventually reliable channels can be built on top of fair-lossy ones in an asynchronous system, S^+ is as strong as $\mathcal{S}^\star$. In S^{++} [3] and [2], every pair of processes is assumed to be connected by fair-lossy channels. Thus, similarly to the above, S^{++} is as strong as $\mathcal{S}^\star$. Finally, the models of [24,32,38] assume that every pair of processes is connected via reliable channels, which again means that these models are strictly stronger than $\mathcal{S}^\star$. Since Ω can be implemented in all models mentioned above [2,3,24,32,38], Corollary 17 implies the required. ◀

7 Related Work

Fault-tolerant distributed computing in the presence of message loss has been extensively studied in the past. Early work focused on the models that, in addition to crashes, allow processes to also fail to either send messages (*send omission*) [28] or both send and receive messages (*generalized omission*) [47]. Both these models have been shown to be equivalent to the crash failures in their computational power in both synchronous [43] and asynchronous [19] systems. They are however too strong to capture partitionable systems, as they would automatically classify any non-crashed process with unreliable connectivity as faulty.

The classical paper by Dolev [20] as well as more recent work (see [49] for survey) study consensus solvability in general networks as a function of the network topology, process failure models, and synchrony constraints. These papers however, only consider the classical (i.e., non-partitionable) version of consensus, which requires agreement to be reached by all correct processes. They also do not consider failure models with flaky channels.

In a seminal paper [50], Santoro and Widmayer proposed a *mobile omission* failure model, which treats message loss independently of process failures. They showed that in order to solve consensus in a synchronous round-based system in the absence of process failures, it is necessary and sufficient to ensure that the communication graph contains a strongly connected component in every round [50, 51]. In contrast, our lower bounds demonstrate that in order to implement consensus (or even a register) in a partially synchronous system, some processes must remain strongly connected throughout the *entire* execution.

Subsequent work explored fault-tolerant computation in the presence of both faulty links and processes under weakened synchrony assumptions. Basu et al. [11] established a separation between reliable channels and either eventually reliable or fair-lossy channels in terms of their power to solve certain problems (such as uniform reliable broadcast and k-set agreement) under asynchrony. However, their notion of a reliable link [29] is too strong to be implementable in practice as it requires every message transmitted via a complete send invocation to be eventually delivered to a correct destination even if the sender later crashes.

Several generalizations of the classical failure detector abstractions of [17] for partitionable systems were proposed in [4,21,22,26] and shown sufficient for consensus in [4,21,26]. However, as we explain in §1, the failure detectors introduced in these papers cannot be implemented in the presence of flaky channels.

Aguilera et al. [3] and follow-up work [2, 24, 32, 38] studied the problem of implementing a failure detector Ω (which is the weakest for consensus [16]), under various weak models of synchrony, link reliability, and connectivity. However, these results are inapplicable in our setting as the models considered in these papers disallow full partitions.

As we show in §6, our connectivity bounds for consensus are also applicable in non-partitionable systems with unreliable channels. In particular, they imply that Ω is not the only factor determining consensus solvability, and that the degree of connectivity being assumed also plays an important role. This suggests an interesting tradeoff across all three modeling dimensions of synchrony, channel reliability, and connectivity considered in this paper. In particular, to solve consensus we assume that every non-faulty channel is eventually timely [23], whereas the algorithm of [2] can cope with weaker synchrony constraints. However, unlike [2], which assumes complete connectivity via fair-lossy channels, our implementation is able to tolerate an adversarial message loss (flakiness). Whether this tradeoff is fundamental, or our synchrony assumptions can be further relaxed remains the subject of future work.

Alquraan et al. [7] present a study of system failures due to network partitions, which we already mentioned in §1. This work highlights the practical importance of coming up with provably correct designs that explicitly consider channel failures. A follow-up paper [6]

introduces a communication layer that can mask channel failures, but only when access to low-level networking infrastructure is available. OmniPaxos [44] ensures liveness of consensus under channel failures, but only handles disconnected channels, not flaky ones.

Raft [45,46] elects leaders in a randomized way: a prospective leader picks a view number and requests *Votes* from a majority of processes (analogous to 1A messages of Paxos). This may lead to split votes when multiple processes are competing to get elected, in which case each process waits for a randomized timeout and retries. While this mechanism works reasonably well in practice, it does not guarantee termination under partial synchrony even with perfect connectivity. Raft also includes a *Pre-Vote* optimization, which, as Jensen et al. point out [33], can help maintain liveness under intermittent connectivity. The optimization requires that before sending *Vote* requests, a prospective leader gathers a majority of *Pre-Votes* from other processes, certifying that they are ready to vote for it. This is similar to gathering WISHes from a majority of processes in our synchronizer before entering a view (Figure 4). But unlike our synchronizer *Pre-Vote* is a best-effort technique, since in between a process granting a *Pre-Vote* and receiving the corresponding *Vote* request it may vote for someone else, thereby creating a split vote. Nevertheless, the presence of techniques similar to ours in existing practical algorithms makes us hopeful that our work could be used to make Paxos-based systems provably live even under adversarial network conditions.

References

1 Yehuda Afek, Hagit Attiya, Alan D. Fekete, Michael J. Fischer, Nancy A. Lynch, Yishay Mansour, Da-Wei Wang, and Lenore D. Zuck. Reliable communication over unreliable channels. *J. ACM*, 41(6):1267–1297, 1994. doi:10.1145/195613.195651.

2 Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Symposium on Principles of Distributed Computing (PODC)*, 2004. doi:10.1145/1011767.1011816.

3 Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega in systems with weak reliability and synchrony assumptions. *Distributed Comput.*, 21(4):285–314, 2008. doi:10.1007/S00446-008-0068-Y.

4 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Sci.*, 220(1):3–30, 1999. doi:10.1016/S0304-3975(98)00235-7.

5 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Comput.*, 13(2):99–125, 2000. doi:10.1007/S004460050070.

6 Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/alfatafta.

7 Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/alquraan.

8 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.

9 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Lower bound on the step complexity of anonymous binary consensus. In *Symposium on Distributed Computing (DISC)*, 2016. doi:10.1007/978-3-662-53426-7_19.

10 Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, 2014. doi:10.1145/2643130.

11 Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Workshop on Distributed Algorithms (WDAG)*, 1996. doi:10.1007/3-540-61769-8_8.

12 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of Byzantine state-machine replication. In *Symposium on Distributed Computing (DISC)*, 2022. doi:10.4230/LIPICS.DISC.2022.12.

13 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live. *Distributed Comput.*, 35(6):503–532, 2022. doi:10.1007/S00446-022-00432-Y.

14 Eric A. Brewer. Towards robust distributed systems (abstract). In *Symposium on Principles of Distributed Computing (PODC)*, 2000. doi:10.1145/343477.343502.

15 Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/brooker.

16 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996. doi:10.1145/234533.234549.

17 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

18 Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001. doi:10.1145/503112.503113.

19 Brian Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Trans. Comput.*, 37(12):1541–1553, 1988. doi:10.1109/12.9732.

20 Danny Dolev. The Byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

21 Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. Technical Report TR96-1608, Department of Computer Science, Cornell University, 1996.

22 Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments (brief announcement). In *Symposium on Principles of Distributed Computing (PODC)*, 1997.

23 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.

24 Antonio Fernández Anta and Michel Raynal. From an intermittent rotating star to a leader. In *Conference on Principles of Distributed Systems (OPODIS)*, 2007. doi:10.1007/978-3-540-77096-1_14.

25 Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998. doi:10.1145/290179.290183.

26 Roy Friedman, Idit Keidar, Dahlia Malkhi, Ken Birman, and Danny Dolev. Deciding in partitionable networks. Technical Report TR95-1554, Department of Computer Science, Cornell University, 1995.

27 Seth Gilbert and Nancy 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.

28 Vassos Hadzilacos. Byzantine agreement under restricted type of failures (not telling the truth is different from telling lies). Technical Report TR-18-63, Department of Computer Science, Harvard University, 1983.

29 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcast and related problems. Technical Report TR94-1425, Department of Computer Science, Cornell University, 1994.

30 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

31 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. doi:10.1109/ICDCS.2003.1203503.

32 Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. Chasing the weakest system model for implementing ω and consensus. *IEEE Trans. Dependable Secur. Comput.*, 6(4):269–281, 2009. doi:10.1109/TDSC.2008.24.

33 Chris Jensen, Heidi Howard, and Richard Mortier. Examining Raft's behaviour during partial network failures. In *Workshop on High Availability and Observability of Cloud Systems (HAOC)*, 2021. doi:10.1145/3447851.3458739.

34 Leslie Lamport. On interprocess communication – Part I: Basic formalism, Part II: Algorithms. *Distributed Comput.*, 1(2):77–101, 1986. doi:10.1007/BF01786227.

35 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

36 Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In *Workshop on Distributed Algorithms (WDAG)*, 1994. doi:10.1007/BFb0020440.

37 Nancy Lynch. *Distributed Algorithms*, chapter 17. Morgan Kaufmann, 1996.

38 Dahlia Malkhi, Florin Oprea, and Lidong Zhou. ω meets paxos: Leader election and stability without eventual timely links. In *Symposium on Distributed Computing (DISC)*, 2005. doi:10.1007/11561927_16.

39 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Comput.*, 11(4):203–213, 1998. doi:10.1007/S004460050050.

40 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *Cryptoeconomics Systems Conference (CES)*, 2020. doi:10.21428/58320208.08912a03.

41 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear Byzantine SMR. In *Symposium on Distributed Computing (DISC)*, 2020. doi:10.4230/LIPICS.DISC.2020.26.

42 Alejandro Naser-Pastoriza, Gregory Chockler, and Alexey Gotsman. Fault-tolerant computing with unreliable channels (extended version), 2023. arXiv:2305.15150.

43 Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *J. Algorithms*, 11(3):374–419, 1990. doi:10.1016/0196-6774(90)90019-B.

44 Harald Ng, Seif Haridi, and Paris Carbone. Omni-Paxos: Breaking the barriers of partial connectivity. In *European Conference on Computer Systems (EuroSys)*, 2023. doi:10.1145/3552326.3587441.

45 Diego Ongaro. *Consensus: bridging theory and practice*. PhD thesis, Stanford University, USA, 2014. URL: https://searchworks.stanford.edu/view/10608105.

46 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

47 Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Software Eng.*, 12(3):477–482, 1986. doi:10.1109/TSE.1986.6312888.

48 Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the PAXOS algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000. doi:10.1016/S0304-3975(00)00042-6.

49 Dimitris Sakavalas and Lewis Tseng. *Network Topology and Fault-Tolerant Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019. doi:10.2200/S00918ED1V01Y201904DCT016.

50 Nicola Santoro and Peter Widmayer. Time is not a healer. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 1989. doi:10.1007/BFB0028994.

51 Nicola Santoro and Peter Widmayer. Distributed function evaluation in the presence of transmission faults. In *Symposium on Algorithms (SIGAL)*, 1990. doi:10.1007/3-540-52921-7_85.

Local Recurrent Problems in the SUPPORTED Model

Akanksha Agrawal
Indian Institute of Technology Madras, India

John Augustine
Indian Institute of Technology Madras, India

David Peleg
Weizmann Institute of Science, Rehovot, Israel

Srikkanth Ramachandran
Indian Institute of Technology Madras, India

Abstract

We study the SUPPORTED model of distributed computing introduced by Schmid and Suomela [27], which generalizes the LOCAL and CONGEST models. In this framework, multiple instances of the same problem, differing from each other by the subnetwork to which they apply. recur over time, and need to be solved efficiently online. To do that, one may rely on an initial preprocessing phase for computing some useful information. This preprocessing phase makes it possible, in some cases, to obtain improved distributed algorithms, overcoming locality-based time lower bounds.

Our main contribution is to expand the class of problems to which the SUPPORTED model applies, by handling also multiple recurring instances of the same problem that differ from each other by some problem specific input, and not only the subnetwork to which they apply. We illustrate this by considering two extended problem classes. The first class, denoted PCS, concerns problems where client nodes of the network need to be served, and each recurring instance applies to some Partial Client Set. The second class, denoted PFO, concerns situations where each recurrent instance of the problem includes a partially fixed output, which needs to be completed to a full consistent solution.

Specifically, we propose some natural recurrent variants of the dominating set problem and the coloring problem that are of interest particularly in the distributed setting. For these problems, we show that information about the topology can be used to overcome locality-based lower bounds. We also categorize the round complexity of Locally Checkable Labellings in the SUPPORTED model for the simple case of paths. Finally we present some interesting open problems and some partial results towards resolving them.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms

Keywords and phrases Distributed Algorithms, LOCAL Model, SUPPORTED Model

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.22

Related Version *Full Version*: `https://arxiv.org/abs/2212.14542`

Funding John Augustine and Srikkanth Ramachandran are supported by the Centre of Excellence in Cryptography, Cybersecurity, and Distributed Trust (CCD) and by an IITM-Accenture project (SB/22-23/007/JOHN/ACC).

Acknowledgements David Peleg holds the Venky Harinarayanan and Anand Rajaraman (VHAR) Visiting Chair Professorship at IIT Madras. This work was carried out in part during mutual visits that were supported by the VHAR Visiting Chair funds.

1 Introduction

1.1 Background and motivation

The area of distributed network algorithms concerns the development and analysis of distributed algorithms operating on a network of processors interconnected by communication links. In particular, a substantial body of research has been dedicated to the development of various graph algorithms for problems whose input consists of the network topology. Examples for such problems are finding a maximal independent set (MIS) for the network, finding a maximal or maximum matching (MM), a minimum dominating set (MDS), a proper coloring with few colors, and so on, and considerable efforts were invested in developing sophisticated and highly efficient algorithms for these problems. Such algorithms are particularly significant in settings where the distributed network at hand is *dynamic*, and its topology keeps changing at a high rate.

The observation motivating the current study is that in many practical settings, the network itself may be static, or change relatively infrequently. In such settings, problems depending solely on the graph structure need be solved only once. In contrast, there are a variety of other problems, related to computational processes that occur repeatedly in the network, which need to be solved at a much higher frequency, and whose input consists of the network topology together with some other (varying) elements. For such problems, the traditional model might not provide a satisfactory solution, in the sense that it may be unnecessarily expensive to solve the entire problem afresh for each instance. Rather, it may be possible to derive improved algorithmic solutions that take advantage of the fact that the network topology is static. We refer to such problems as *recurrent problems*.

We envision that depending on the desired problems that the network needs to support, one can compute and store additional information about the topology of the network within each node, to enable recurrent problems to be solved faster. Inherently this captures an aspect of network design. When a network is built, it maybe useful to compute useful information about its topology keeping in mind the recurrent problems that it must support during its lifetime.

This framework has been studied in the literature as the SUPPORTED model [27]. However, the recurrent problems studied so far within this framework were mostly limited to instances of the original problem defined on an edge induced subgraph of the original graph[1].

We believe that this limited class of problems does not fully capture all recurrent problems of interest that can arise in the SUPPORTED model. To demonstrate this, we introduce two new classes of recurring problems that can occur in the SUPPORTED model, namely, problems with *partial client set (PCS)* and problems with *partially fixed output (PFO)*, defined below. We illustrate these classes by focusing on natural extensions of the classical local problems of coloring and dominating set, and developing algorithms and lower bounds for them in the SUPPORTED model.

1.2 Recurrent Problems

We consider graph-related optimization problems each of whose *instances* $\langle G, \mathsf{S} \rangle$ consists of a network topology $G = (V, E)$, on which the distributed algorithm is to be run, and some problem-specific input S. The term "recurrent problem" refers to a setting where the

[1] One recent exception [19] introduced a different type of recurrent problem concerning matrix multiplication, where the structure of the input matrices is fixed but the specific values of the nonzero elements are different in each recurring instance. The problem is studied in the (supported) node-congested clique model of distributed computing.

network G is fixed, and is the same in all instances (hence we often omit it). Formally, there is a stream of instances that arrive from time to time and must be solved efficiently. The optimization problem itself may be a variant of some classical graph optimization problem, except it has some additional restrictions, specified in each instance S. Two concrete types of restrictions that are of particular interest are *partial client set (PCS)* and *partially fixed output (PFO)*.

Partial client set (PCS)

An instance S restricted in this manner specifies a subset $C \subseteq V$ of *client vertices* to which the problem applies. The rest of the vertices are not involved (except in their capacity as part of the network). For example, consider the maximum matching problem. In the PCS variant of this problem, a PCS-restricted instance will specify a vertex subset C such that the matching is only allowed (and required) to connect vertex pairs of C.

Partially fixed output (PFO)

An instance S restricted in this manner specifies a part of the output. The rest of the output must be determined by the algorithm. For example, consider the k-centers problem (where the goal is to select a subset C of k vertices serving as centers, so as to minimize the maximum distance from any vertex of V to C). In the PFO variant of the k-centers problem, a PFO-restricted instance will specify a vertex subset C_{pre} of k' vertices that were already pre-selected as centers, and the task left to the algorithm is to select the remaining $k - k'$ centers.

Naturally, some recurrent problems may involve other novel restrictions as well as hybrids, thereby opening up the possibility for rich theory to be developed.

1.2.1 Two representative examples: CDS and PCC

In this paper, we will focus on two concrete examples for recurrent problems of practical significance, and use them for illustration. The first of these two example problems, named CDS, serves to illustrate a recurrent problem with PCS-restricted instances (where the set of clients changes in each instance). The second problem, named CC, illustrates a recurrent problem with PFO-restricted instances (where parts of the output are fixed in advance in each instance).

Minimum client-dominating set (CDS)

In certain contexts, a dominating set D in a network G (i.e., such that every vertex $v \in V$ either belongs to D or has a neighbor in D) is used for placing *servers* providing some service to all the vertices in the network (interpreted as *clients*), in settings where it is required that each vertex is served by a server located either locally or at one of its neighbors. The *minimum dominating set (MDS)* problem requires finding the smallest possible dominating set for G.

We consider the recurrent variant of the CDS problem with PCS-restricted instances. This problem arises in settings where the set of clients in need of service does not include all the vertices of G, but rather varies from one instance to the next. In such settings, the network G is static, and from time to time, a set of clients $C \subseteq V$, formed in an ad-hoc manner due to incoming user requests, requests to select and establish a (preferably small) subset D of vertices from among their neighbors, which will provide them some service. In other words, the set D is required to dominate the vertices in C. On the face of it, solving the minimum dominating set problem once on G may be useful, but not guarantee optimal

results for each recurrent instance S; rather, for each instance S, it may be necessary to solve the specialized problem once the request appears in the network. Hereafter, we refer to this problem as *minimum client-dominating set (*MCDS*)*.

Note that one may also consider a generalized problem that includes also a PFO component, by specifying in each instance S also a partial set D' of vertices that were pre-selected as servers (or dominators). Our results are presented for the MCDS problem (without PFO restrictions), but it should be clear that they can easily be extended to the generalized problem with PFO restrictions[2].

Color Completion (CC)

In certain contexts, a proper coloring of a distributed network is used for purposes of scheduling various mutually exclusive tasks over the processors of the network. For example, suppose that performing a certain task by a processor requires it to temporarily lock all its adjacent links for its exclusive use, preventing their use by the processor at the other end. Employing a proper coloring as the schedule (in which all the processors colored by color t operate simultaneously at round t) enables such mutually exclusive operation. Naturally, it is desirable to use as few colors as possible, in order to maximize parallelism.

We consider the recurrent variant of the coloring problem with PFO-restricted instances. From time to time we may receive a partial (collision-free) coloring assignment to some subset $C \subseteq V$ of the vertices, representing processors constrained to operate on some particular time slots. We are required to color all the remaining vertices in $V \setminus C$ properly and consistently with the initial coloring. Typically, using colors already occurring in the precoloring (i.e., used by some vertices in the set C) is fine, since these time slots are already committed for the task at hand. However, it is desirable to use as few new time slots (or new colors), to minimize the overall time spent on the task.

Note that one may also consider a generalized problem that includes also a PCS component, by specifying in each instance S also a partial set V' of vertices that are interested in being scheduled, and hence need to be colored. Our results are presented for the CC problem (without PCS restrictions), but it should be clear that they can easily be extended to the generalized problem with PCS restrictions[3].

1.3 The SUPPORTED model

The SUPPORTED model is an extension of the well studied LOCAL and CONGEST models with an additional preprocessing phase. Specifically the solution to a problem in the SUPPORTED model consists of two stages, (i) a *preprocessing* stage and (ii) an *online* stage.

- In the preprocessing stage, run an algorithm $\mathcal{A}_{\text{pre}}(G)$ on the topology of the network G and obtain information $\mathsf{Inf}(G)$ to be stored at the network vertices (different vertices may of course store different information).
- During runtime, a stream of instances arrive. Whenever a new instance S arrives, run a distributed algorithm $\mathcal{A}(\mathsf{S}, \mathsf{Inf}(G))$ to solve this problem instance.

In view of the fact that the preprocessing stage takes place only once, the particulars of the preprocessing algorithm are less important to us, and we allow it to be arbitrary (even oracular). For the scope of this paper, in the upper bounds that we show, all our preprocessing phases are explicitly computable, whereas the lower bounds hold for any arbitrary preprocessing.

---

[2] Essentially, for this problem, the pre-selected vertices of D' can be used to satisfy all the clients that neighbor them, leaving us with a smaller set C' of unsatisfied clients that need to be covered.

[3] Essentially, for this problem, the vertices of $V \setminus V'$, which do not require coloring, can simply avoid participating in the coloring process.

In the online stage, we insist that the computation performed by each node in a single round must be polynomial in the size of the graph. Therefore even knowledge of the complete network for each node might not be sufficient, as underlying information about the topology (such as chromatic number) might not be computable in polynomial time.

For a given problem Π on a graph G, one may seek to optimize several parameters. For the scope of this paper, we consider only two, (i) the round complexity of the online algorithm, i.e., the number of synchronous rounds required to solve each recurrent instance and (ii) the size of the output to each node in the preprocessing phase, i.e., the amount of additional information that needs to be stored in each node of the graph from the preprocessing phase.

1.4 Our Contributions

Client Dominating Set (Section 2). We first show that even on a path of n nodes, it is not possible to (i) optimally solve CDS in $o(n)$ time and (ii) compute a $1+\epsilon$ approximation in $\Omega(\epsilon^{-1})$ rounds (whenever $\epsilon = \Omega(1/D)$). On the other hand, we show that for trees and planar graphs, one can obtain a $1+\epsilon$ approximation in $O(\epsilon^{-1})$ and $O(\text{poly}(\epsilon^{-1}))$ rounds respectively. The algorithm for trees decomposes it into blocks of depth $\Theta(\epsilon^{-1})$ and executes an optimal algorithm within each block. The algorithm for planar graphs is an adaptation of the $O(\text{poly}(\epsilon^{-1})\log^* n)$ round LOCAL algorithm of [14]. We look at the most time consuming part of their algorithm and we speedup a particular sub-routine. In order to do that, we make use of a combinatorial object called *non-repetitive* coloring, first proposed by [2] who conjectured that it is bounded in planar graphs and recently [15] showed an upper bound of 768. To the best of our knowledge, this is the first application of such a coloring in the distributed setting.

Color Completion (Section 3). We provide an algorithm to complete a given coloring using at most $\chi(\Delta+1)/k$ new colors in k rounds, for any $1 \leq k \leq \chi$. We show that for $k=1$, the number of colors used is asymptotically tight in the worst case. Tighter analysis of the same algorithm when $k=\chi$ shows that only χ new colors are needed when $k=\chi$.

Locally Checkable Labellings (Section 4). We study a generic class of problems called Locally Checkable Labellings (LCL) [26]. We show that on a path, every LCL problem either has worst case complexity $O(1)$ or $\Theta(n)$. In the specific case of recurrent problems where the online instances are a specific LCL on a sub-path of the given path, we provide an efficient centralized algorithm to classify the LCL into one of the two cases and also construct the distributed algorithm to solve an LCL given its description.

Miscellaneous Problems (Section 5). Finally, we provide some partial results on sub-graph maximal matching and sub-graph maximal independent set that could potentially be useful in finding optimal solutions for these problems in the SUPPORTED model.

1.5 Related Work

The SUPPORTED model was first proposed by [27]. [16] provide several results including lower bounds for problems such as sinkless orientation and approximating independent set. [20] provide near optimal algorithms for global network optimization problems, such as minimum spanning tree, min-cut etc.

Dominating Set. [14] provided an $O(\text{poly}(\epsilon^{-1})\log^* n)$ round algorithm for a $1+\epsilon$ approximation for the dominating set problem and it was later extended to bounded genus graphs by [3]. [16] briefly discuss about extending these results to the SUPPORTED model.

Coloring. Color Completion has been one of the methods used for $\Delta + 1$ coloring graphs in $\log^* n + f(\Delta)$ rounds. Existing algorithms decide on a coloring for a sub-graph of the given graph and then recursively complete the chosen coloring. [7] provided the first sub-linear in Δ algorithm. The current best known algorithm has round complexity $\log^* n + O(\sqrt{\Delta \log \Delta})$ (see [24, 8, 17]). [24] also provided a smooth tradeoff between the number of colors and the round complexity, specifically in $k + \log^* n$ rounds, graphs can be properly colored using $O(\Delta^2/k^2)$ colors for any $1 \leq k \leq \sqrt{\Delta}$. We note that Maus's algorithm ([24]) does not provide a $\Delta + 1$ coloring but rather an $O(\Delta)$ coloring.

LCL. Locally Checkable Labellings (LCL) were first proposed by [26]. [11] showed gaps in the deterministic complexity of LCL's. They showed that the worst case deterministic round complexity of LCL's on any hereditary graph class is either $\omega(\log_\Delta n)$ or $O(\log^* n)$. They also show that for paths, there is no LCL with complexity $o(n)$ and $\omega(\log^* n)$. Later [12] and [18] showed that on trees, the deterministic worst case complexities for LCL's is either $O(1), \Theta(\log^* n), \Theta(\log n)$ or $n^{\Theta(1)}$. More recently, [6] showed that for a more restricted class of LCL problems, on rooted trees, there is a centralized algorithm that takes as input the description of the LCL and decides which of the above complexity classes it belongs to. Given the LCL, deciding its distributed complexity class on trees was shown to be EXPTIME hard by [10].

[13] studied the complexity of LCL's on unlabelled paths and cycles. For this easier case, they showed the correspondence between a solution to the LCL and a non-deterministic finite automaton. This observation helps to identify polynomial-time computable properties of the LCL which almost exactly determine its worst round complexity, with the exception of a case that turns out to be co-NP complete. Our characterisation of LCL problems in SUPPORTED in Section 4 follows along the same ideas. [9] study the round complexity of LCL problems in two dimensional grid and toroids. Even though these topologies are simple extensions of paths and cycles, and the complexity classes of LCL problems are exactly the same $(O(1), \Theta(\log^* n), \Theta(n))$, characterizing the round complexity is undecidable.

2 Dominating Sets

2.1 Client Dominating Set

▶ **Definition 1** (Client Dominating Set (CDS)). *Given a graph G and a subset of its vertices $C \subseteq V(G)$, called the client set, we say that a set $D \subseteq V(G)$ dominates C if for every client $c \in C$, there exists $v \in D$ such that either $v = c$ or v is a neighbor of c. D is said to be a client dominating set of G for the clients C.*

The minimum client dominating set problem (MCDS) asks for a CDS of minimum size. The CDS problem is of course a generalization of the classical Dominating Set problem as the dominating set is precisely the case when $C = V(G)$. It is also possible to reduce the CDS problem to an instance of the Dominating Set problem. See the full paper [1] for a discussion of the reductions. Our reduction does not preserve *locality* and so it does not provide any insight into the round complexity of this problem in distributed settings.

2.2 Lower Bound for Paths

We establish two lower bounds for MCDS on a path. First, we argue that, regardless of the preprocessing, the online runtime of every (exact) deterministic distributed algorithm for the MCDS problem must take time $\Omega(D)$ on networks of diameter D. Second, we show that the

online runtime of every deterministic distributed approximation algorithm for MCDS with ratio $1+\epsilon$ must require time $\Omega(1/\epsilon)$ on some input. The proofs of these theorems appear in the full paper [1].

▶ **Theorem 2.** *Let $\mathcal{A}$ be a deterministic distributed local algorithm for* CDS *with arbitrary preprocessing. Then there exists some input for which $\mathcal{A}$ requires $\Omega(D)$ time.*

▶ **Theorem 3.** *Let $\mathcal{A}$ be a deterministic distributed local approximation algorithm for* CDS*, with arbitrary preprocessing, whose online runtime on every path and every instance is at most $k = 4\ell+1$ for some integer $\ell \geq 1$. There exists a network and a set of clients for which the approximation ratio of $\mathcal{A}$ is at least $1+1/(k+2)$.*

2.3 A CTAS for Trees

In this section we describe a *constant time approximation scheme* (CTAS) for MCDS on trees, i.e., a $1+\epsilon$ approximation in $O(\epsilon^{-1})$ rounds.

The algorithm for trees is based on a preprocessing stage in which the tree is partitioned into subtrees of depth $O(k)$ for some integer parameter k. Each recurrent instance is then solved by computing a local near-optimal CDS on each subtree, while taking care to ensure that the resulting solutions combine into a $1+4/(k-1)$ approximation of the optimal global solution. The "interface" between adjacent subtrees is more difficult to handle, as making a single change in the selection in one subtree (e.g., in one of its leaves) might affect several adjacent subtrees, which makes both the algorithm and its analysis somewhat more complex.

Let us first describe the preprocessing stage, which is applied to the network tree T. The algorithm has an integer parameter $\ell \geq 1$ and sets $k = 4\ell+1$. Root the tree T at a root vertex r_0, and mark each vertex v by its layer, $layer(v)$, namely, its distance from r_0 (by definition $layer(r_0) = 0$). Partition the tree T into subtrees by taking every vertex v with $layer(v) = pk$ for integer $p \geq 0$ as a root and defining $T[v]$ as the subtree of depth k rooted at v. See Fig. 1(a). For notational convenience, we sometimes use $T[v]$ to denote also the *vertex set* of the subtree $T[v]$. Also, for any subtree $T[v]$ and vertex set $X \subseteq T$, we denote $X[v] = X \cap T[v]$.

The leaves of a subtree $T[v]$ can be classified into *real leaves* and *layer-leaves*, namely, leaves of $T[v]$ that are internal nodes in T. A subtree that has no other subtree below it (namely, all of whose leaves are real) is called a *leaf-subtree* or simply L-tree. Otherwise, it is called an *internal-subtree* or I-tree. (See Fig. 1(a).) We partition the vertices of T into two subsets. Let $lleaves$ be the set of all layer-leaves, and int be the set of all remainig vertices. This induces a partition of the vertices of each subtree $T[v]$ into $lleaves[v]$ and $int[v]$. (For an L-tree, $lleaves[v] = \emptyset$.)

During the recurrent stage, each instance consists of a set S of clients. This induces additional distinctions on the tree structure. Internal subtrees are classified into two types. The I-tree $T[v]$ is called a *cut I-tree* if on every path from v to a root w hanging from a layer-leaf of $T[v]$ there are two consecutive vertices that do not belong to S. See Fig. 1(b). The figure also illustrates the fact that in a cut I-tree $T[v]$ one can identify a subtree $\check{T}[v]$, referred to as the *peak* of $T[v]$, the minimal sub-tree with the property that for every edge (x,y) connecting a vertex $x \in \check{T}[v]$ to a child $y \notin \check{T}[v]$, both $x, y \notin S$. This implies that nodes *below* $\check{T}[v]$ cannot help in dominating clients in $\check{T}[v]$, namely, taking them into D cannot dominate client vertices in $\check{T}[v]$. $T[v]$ is a *full I-tree* if it is not a cut I-tree, namely, there is at least one path from v to a root w hanging from some layer-leaf of $T[v]$ with no two consecutive vertices of $V \setminus S$.

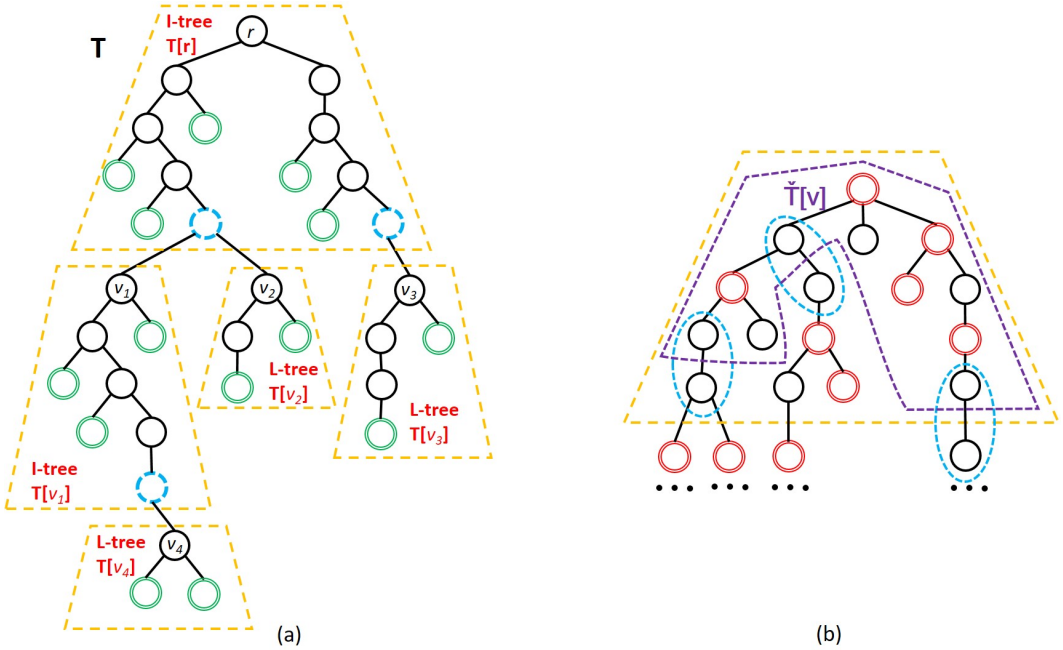


Figure 1 (a) A decomposition of the tree T into subtrees for $\ell = 1$, $k = 5$. Layer-leaves are marked by a blue dashed circle, and real leaves are marked by a green double circle. (b) A cut I-tree, $k = 5$. The client vertices of S are drawn as double red circles. The cuts along root-to-root paths are marked by blue dashed ellipses. The peak-tree $\check{T}[v]$ is marked by a purple dashed curve.

The idea behind the approximation scheme is as follows. Our algorithm solves the CDS problem *separately*, in an optimal manner, on each subtree $T[v]$ of depth at most k for the client set $S[v]$. This can be done in time $O(k)$, but might entail inaccuracies. As illustrated in the lower bound of Sect. 2.2, the main hindrance to the accuracy of a local distributed algorithm for MCDS stems from long paths with a periodic occurrence of client vertices. Such a path, threading its way along some root-to-leaf path in T, might be handled poorly by the local computations. Our goal is to bound the loss by at most 1 per subtree in the decomposition. This is justified for full I-trees, since in a full I-tree the optimum solution D^* must also use $\Omega(k)$ dominators to cover all the clients, so the inaccuracy ratio is just $1/\Omega(k)$.

This approach is made complicated due to the fact that some subtrees are not full, and may require only a small number of dominators. For such subtrees (specifically, L-trees and cut I-trees), we cannot allow the algorithm to "waste" more than the optimum solution. Hence when comparing the number of dominators used by the algorithm to that of the optimum D^*, we must use an accounting method that will equate the costs over L-trees and cut I-trees, and charge all the "waste" to full I-trees.

This is done as follows. In a first phase, we locally solve the problem optimally in each L-tree and cut I-tree, i.e., dominate clients in these sub-trees using only vertices of these sub-trees. This is only used in order to decide, for each such subtree $T[v]$, whether the root's parent, denoted $\mathsf{parent}(v)$, must belong to the dominating set. This is important since these vertices cover the "interference layers" between adjacent subtrees. For the full I-trees, an optimal local solution cannot be computed. Therefore, we artificially impose a "waste" in every full I-tree $T[v]$, by selecting the parent of its root, $\mathsf{parent}(v)$, as a dominator, whether or not necessary. As explained above, this "waste" is justified by the fact that D^* must also use

$\Omega(k)$ dominators in these subtrees. As a result, when we compute a dominating set for the remaining undominated clients in the second phase of the algorithm, the solution computed by the algorithm on each subtree T' is no greater than the number of D^* dominators in T'.

Optimal procedure P_{up}

A simple procedure P_{up} we use is an optimal algorithm for CDS on rooted trees, which runs in time $O(\mathsf{depth}(T))$ on a tree T. The algorithm starts with an empty set of dominators D and works its way from the leaves up, handling each node w only after it finishes handling all its children. It adds w to the set D in one of the following two cases:

(1) Some of w's children are clients and are not yet dominated, or

(2) w itself is an undominated client and is the root.

It is easy to verify that this algorithm yields a minimum cardinality solution for CDS. It is also easy to implement this greedy algorithm as an $O(\mathsf{depth}(T))$ time distributed protocol.

Modification for subtrees. When applying this procedure to a subtree $T[v]$ of T where v is not the root of T, we make the following small but important modification: When the procedure reaches v itself, if $v \in S$ and v is still non-dominated, then we add $\mathsf{parent}(v)$ instead of v to the solution. (This can be done since $\mathsf{parent}(v)$ belongs to the tree T, although it is outside the subtree $T[v]$.)

The above modification is enough to obtain an approximation scheme. For complete pseudocode and analysis, see the full paper [1]. We get the following result.

▶ **Theorem 4.** *For every positive integer k, there exists a deterministic distributed local approximation algorithm for* CDS*, with preprocessing allowed, whose online runtime on every n-vertex tree and every instance is at most $O(k)$ with approximation ratio of at most $1+\frac{4}{k-1}$.*

2.4 A CTAS for MCDS on Planar Graphs

2.4.1 Constant Approximation for MCDS on Planar Graphs

The state of the art algorithm for constant round planar dominating set approximation in the LOCAL model achieves an approximation ratio of 20 by a recent work of [21]. Their algorithm and analysis extend to the client dominating set problem with slight modifications. See Algorithm 1 for the pseudocode.

Algorithm 1 Constant Approximation for MCDS in Planar Graphs.

1: $C \leftarrow$ client set
2: For every $A \subseteq V(G)$, define $N_C(A) = \{w \mid w \in C \text{ and } (w,v) \in E(G) \text{ for some } v \in A\}$
3: $N_C[A] = N_C(A) \cup (A \cap C)$
4: $D_1 \leftarrow \{v \in V(G) \mid \forall A \subseteq V(G) \setminus \{v\}, N_C[A] \supseteq N_C(v) \Rightarrow |A| \geq 4\}$
5: For every $v \in V(G)$, compute $B_v = \{w \in V(G) \setminus D_1 \mid |N_C(v) \cup N_C(w)| \geq 10\}$
6: $D_2 \leftarrow \{v \in V(G) \setminus D_1 \mid B_v \neq \emptyset\}$
7: $D_3 \leftarrow C \setminus N_C[D_1 \cup D_2]$
8: Return $D_1 \cup D_2 \cup D_3$

▶ **Theorem 5.** *Algorithm 1 provides a 39-approximation for the* MCDS *problem in planar graphs.*

2.4.2 A $1+\epsilon$ approximation

We adapt the distributed $1+\epsilon$-approximation scheme of [14], whose round complexity is $O((\frac{1}{\epsilon})^c \log^* n)$ where $c = \log_{24/23} 3$.

The only non-constant round part of their algorithm is in 3-coloring several pseudo-forests[4] that are obtained as follows. A partition of the vertices of the graph is computed such that each part induces a connected component of diameter $O(\text{poly}(\epsilon^{-1}))$. Each component is then contracted into a single node and a pseudo-forest of the resulting graph is computed by a greedy procedure. The choice of the pseudo-forest and the partition is dictated by the clients, and thus hard to predict without knowledge of the clients.

We first describe the major changes required to adapt this method to the CDS problem. We then discuss the preprocessing that helps to 3-color the pseudo-forests in $O(\text{poly}(\epsilon^{-1}))$ rounds, thus removing the $\log^* n$ factor.

Adapting to CDS. First, we remove the edges that are not incident on any client. These edges do not contribute to the criteria for a set to be a dominating set, and they can be ignored. We then compute a constant approximation $\tilde{D}$ as per Algorithm 1. The initial clustering is obtained by choosing for each client c an arbitrary dominator of c from $\tilde{D}$ and contracting the edge between them. Additionally, every vertex that is neither a client nor a dominator chooses an arbitrary neighboring client and the edge between them is merged. The remaining steps are identical to the previous procedure.

Speeding up using a preprocessing phase. One natural candidate to consider for the preprocessing stage is a proper 4-coloring of the planar graph. Unfortunately, while a coloring of any graph remains valid after the removal of edges or vertices, it does not remain valid after contractions. An arbitrary precomputed coloring might not be of much use in coloring the contracted graphs that arise from repeated contractions. To accommodate contractions, we precompute a *non-repetitive* coloring of G (which is the only output of our preprocessing phase). A *non-repetitive* coloring is a coloring of the graph such that for any odd length simple path, the ordered set of colors in the first half of the path is different from that of the second half. Non-repetitive colorings were first proposed by [2]. The minimum number of colors required to realise a non-repetitive coloring is called the Thue number of the graph and is denoted by $\pi(G)$. [15] showed recently that $\pi(G) \leq 768$ for all planar graphs G.

Suppose we have a pseudo forest F that needs to be 3-colored and suppose F is obtained from G_t, the contracted graph. Let $\mathsf{out}(v)$ denote the other end of the outgoing edge of v in F. In order to 3-color the forest, it is sufficient to choose colors in such a way that $\mathsf{out}(v)$ and v have different colors, for every v. We can associate with each node v of G_t, a connected component (denoted G_v) in the original graph G that contains the ends of all edges that were contracted to v. Choose any edge e that crosses G_v and $G_{\mathsf{out}(v)}$. Construct a spanning tree of G_v and root it at the endpoint $r(v)$ of e that lies in G_v. We now color v with the ordered set of non-repetitive colors traced on the unique path from $r(v)$ to $r(\mathsf{out}(v))$, excluding $r(\mathsf{out}(v))$, in the graph $G_v \cup G_{\mathsf{out}(v)} \cup \{e\}$. We enumerate these colors from 1 to 768^{d+1} where d is the maximum diameter of the clusters. Let the computed path be P_v. Observe that whenever $\mathsf{out}(\mathsf{out}(v)) \neq v$, the paths P_v and $P_{\mathsf{out}(v)}$ can be concatenated to form a simple path in the graph G. If P_v and $P_{\mathsf{out}(v)}$ have different lengths, then the colors assigned to them are different. Otherwise, by the property of a non-repetitive coloring, the

[4] A pseudo-forest is a directed graph wherein every node has exactly one outgoing edge.

ordered set of colors of P_v and $P_{\mathsf{out}(v)}$ must be different. When $\mathsf{out}(\mathsf{out}(v)) = v$, we have a 2-cycle. In this case we color one of the nodes $\{v, \mathsf{out}(v)\}$ (whichever has higher id, say v) with its own non-repetitive color and redefine $P_v = \{\mathsf{out}(v)\}$. Now the paths P_v and $P_{\mathsf{out}(v)}$ may be concatenated to obtain a simple path P. See Algorithm 2 for the pseudo-code.

Algorithm 2 3-coloring pseudo-forest.

procedure 3-COLOR
Input:
(i) $\mathsf{color} : V(G) \to [768]$, A non-repetitve coloring of the given planar graph G
(ii) $\mathsf{cluster} : V(G) \to \mathbb{N}$, describes a partitioning of the vertices of G that induce connected components of diameter at most d
(iii) G_t : the graph where every cluster is contracted to a single node.
(iv) $\mathsf{out} : V(G_t) \to V(G_t)$, describes a pseudoforest in the graph G_t ▷ $\mathsf{out}(v)$ is the other end of the unique outgoing edge from v
Output: $\mathsf{color}_f : V(G_t) \to [3]$, a proper 3-coloring of the given pseudoforest
  for all clusters $v \in V(G_t)$ (in parallel) **do**
    $p \leftarrow \mathsf{out}(v)$, the parent of v in pseudo-forest of G_t
    Let G_v, G_p be the connected components of G that are contracted to v, p in G_t
    $e_v \leftarrow$ any edge in G that crosses G_v, G_p and $r_v \leftarrow$ the end of e in G_v
    $T_v \leftarrow$ Any spanning tree of G_v, rooted at r_v
  end for
  for all clusters $v \in V(G_t)$ (in parallel) **do**
    if $\mathsf{out}(p) \neq v$ or $v < p$ **then** ▷ detect cycles of length 2
      $\mathsf{path}(v) \leftarrow$ The unique path from r_v to r_p in the graph $T_v \cup T_p \cup \{e\}$
    else ▷ Treating the case of cycle of length 2 separately
      $\mathsf{path}(v) \leftarrow \{r_v\}$
    end if
  end for
  $\mathsf{color}_f(v) \leftarrow$ the ordered set of colors in $\mathsf{path}(v)$
  Enumerate $\mathsf{color}_f(v)$ using integers from 1 to 768^{d+1}
  Reduce $\mathsf{color}_f(c)$ to a 3-coloring using the Cole-Vishkin Algorithm
  return color_f
end procedure

We now have a 768^{d+1} coloring of the pseudo-forest F, which can then be reduced to a 3-coloring using the Cole-Vishkin Algorithm. The complexity is $O(d \log^* 768^{d+1}) = O(d \log^* d)$. This leads us to our main lemma:

▶ **Lemma 6.** *Given a clustering of G into connected components of diameter d, let G' denote the graph obtained after contracting each cluster into a single vertex, and let H be a given pseudo-forest subgraph of G', Algorithm 2 provides a 3-coloring of H in $O(d \log^* d)$ rounds.*

Algorithm 2 is the main unique ingredient to our adaptation of [14]'s algorithm. Plugging this component into their algorithm directly leads to an $O(\mathrm{poly}(\epsilon^{-1}))$ round algorithm. For concreteness, the complete clustering procedure is described in Algorithm 3 with some minor changes to account for the clients. Once the clustering is done, we proceed in the same way, i.e., solve the CDS problem optimally and independently within each cluster. Solving CDS exactly requires NP-Hard problems to be solved in the online phase, which may be undesirable. This can be fixed by replacing the optimal solution with a PTAS in planar graphs for the CDS problem by a similar adaptation of Baker's algorithm [4].

Algorithm 3 Clustering for Planar CDS.

Input: Client set C, a non-repetitive coloring of G and ϵ.
Output: A $1+\epsilon$ approximation of the optimal set dominating C.
Phase 1: Finding a good initial clustering.
Remove all edges that do not have a client incident on them.
Remove isolated vertices after previous step.
Compute a constant approximation $D^\star$ for C using Algorithm 1.
for all nodes $v \in V(G) \setminus D^\star$ **do**
  if v has a neighbor in $D^\star$ **then**
    $u \leftarrow$ any neighbor in $D^\star$
  else
    $u \leftarrow$ any neighbor in C or $\perp$ (if such a node doesn't exist)
  end if
  Contract the edge $e = (u, v)$, if u exists
end for
  ▷ Done in parallel and implicitly, i.e., contracted vertices know their neighbors
Phase 2: Improving the clustering
$G_0 \leftarrow$ underlying simple graph obtained at end of Phase 1.
Set $\mathsf{wt}(e) \leftarrow 1$ for all $e \in E(G_0)$
for all $t = 0, 1, \ldots \lceil \log_{24/23} \frac{234}{\epsilon} \rceil$ **do**
  $\mathsf{out}(u) \leftarrow$ any neighbor v such that $\mathsf{wt}((u, v))$ is maximized
  $H \leftarrow$ induced by the edges $\{(\mathsf{out}(u), u) \mid u \in G_t\}$ ▷ Heavy pseudo-forest
  $\mathsf{col} \leftarrow$ 3-coloring of H obtained using Algorithm 2.
  for all $u \in H$ with $\mathsf{col}(u) = 1$ (in parallel) **do**
    $I_u, O_u \leftarrow \{(u, v) \mid u = \mathsf{out}(v)\}, \{(u, v) \mid v = \mathsf{out}(u)\}$
    Remove either I_u or O_u from H, whichever has smaller total weight
  end for
  for all $u \in H$ with $\mathsf{col}(u) = 2$ (in parallel) **do**
    $I_u, O_u \leftarrow \{(u, v) \mid u = \mathsf{out}(v), \mathsf{col}(v) = 3\}, \{(u, v) \mid v = \mathsf{out}(u), \mathsf{col}(v) = 3\}$
    Remove either I_u or O_u from H, whichever has smaller total weight
  end for
    ▷ H now consists of connected components with diameter at most 10.
  $F \leftarrow$ rooted spanning forest of H
  $E_F, O_F \leftarrow$ edges of F at even and odd depths respectively
  Remove either E_F or O_F, whichever has smaller total weight
  For all edges $e \in E(H)$, contract e in G_t
  $G_{t+1} \leftarrow$ underlying simple graph obtained after contractions.
  For all edges $e = (u, v) \in G_{t+1}$, set $\mathsf{wt}(e) \leftarrow$ number of edges between u, v after all contractions of edges in H.
end for
return G_T

▶ **Theorem 7.** *For every planar graph G, $1+\epsilon$ approximation of* MCDS *can be obtained in $O(\epsilon^{-c} \log^* \epsilon^{-1})$* SUPPORTED *rounds, where $c = \log_{24/23} 3$, using only $O(1)$ additional bits stored in each node as the output of the preprocessing phase.*

The complete pseudocode and proof of completeness appear in the full paper [1].

3 Color Completion

Consider a graph $G(V, E)$ and a coloring $c : V \mapsto \{1, \ldots, k\}$. The vertex v is *properly colored* if each of its neighbors has a different color. The classical vertex coloring problem requires deciding if there exists a coloring for which all vertices are properly colored. When some of the vertices are already assigned a predefined coloring, the resulting recurrent problem is referred to as *color completion* (CC). We use the following measures for evaluating the number of colors used in any valid solution.

- Let $\mathcal{P}_{pc}$ be the set of colors used by the precolored vertices, and denote $\chi_{pc} = |\mathcal{P}_{pc}|$.
- Let $\mathcal{P}_{un}$ be the set of colors used for the uncolored vertices; denote $\chi_{un} = |\mathcal{P}_{un}|$.
- Let $\mathcal{P}_{new} = \mathcal{P}_{un} \setminus \mathcal{P}_{pc}$ be the *new* colors used for the uncolored vertices; denote $\chi_{new} = |\mathcal{P}_{new}|$.
- Let $\mathcal{P}_{all} = \mathcal{P}_{pc} \cup \mathcal{P}_{new}$ be the final set of colors of all vertices; denote $\chi_{all} = |\mathcal{P}_{all}|$.

For a given instance of CC, let χ^*_{un} (respectively, χ^*_{new}, χ^*_{all}) be the smallest possible value of χ_{un} (resp., χ_{new}, χ_{all}) over all possible proper color completions of the precoloring. Additionally, for a given algorithm $\mathcal{A}$, let $\chi^{\mathcal{A}}_{un}$ (respectively, $\chi^{\mathcal{A}}_{new}$, $\chi^{\mathcal{A}}_{all}$) be the value of χ_{un} (resp., χ_{new}, χ_{all}) in the solution computed by $\mathcal{A}$ for the instance.

The efficiency of an algorithm for CC can be measured by two parameters of interest, namely, χ_{new} and χ_{all}. The difference between them becomes noticeable in instances where the colors in $\mathcal{P}_{pc}$ are not contiguous, i.e., when there are colors x such that $x \notin \mathcal{P}_{pc}$ but $x + 1 \in \mathcal{P}_{pc}$. We denote by $\mathsf{CC}_{new}(t)$ (resp. $\mathsf{CC}_{all}(t)$) the problem of color completion such that χ_{new} (resp. χ_{all}) is at most t.

3.1 Single Round Color Completion

We first consider what can be done when the online algorithm is restricted to a single round of communication.

▶ **Theorem 8.** *Consider a graph G with maximum degree $\Delta = \Delta(G)$ and chromatic number $\chi = \chi(G)$ with $\Delta > 0$, then $\mathsf{CC}_{new}(\chi \cdot \Delta)$ can be solved in a single* SUPPORTED *round.*

Proof. The algorithm uses the color palette

$$\mathcal{P} \;=\; \{(i, j) \mid 1 \le i \le \chi, \quad 1 \le j \le \Delta\}.$$

In the preprocessing stage, compute a proper *default coloring* dc of the graph using the color palette $\mathcal{P}^{def} = \{i \mid 1 \le i \le \chi\}$, and let each vertex v store its default color $dc(v)$ for future use. These values are not used as colors in the final coloring.

In the online stage, we are given an arbitrary precoloring $c(w) \in \mathcal{P}$ for some nodes, and need to complete it to a proper coloring by selecting a color $c(v)$ for each non-precolored node v. (It is assumed that the precoloring itself is proper, i.e., no two precolored neighboring vertices use the same color.)

The algorithm requires a single round of communication. Each precolored node w informs its neighbors about its color $c(w)$. Now consider a non-precolored node v. If all neighbors of v are colored, then v chooses a free color from the color palette. As $\chi \cdot \Delta \ge 2\Delta \ge \Delta + 1$, such a color is guaranteed to exist.

Otherwise, v finds a *free color* of the form $(dc(v), j)$ for $1 \le j \le \Delta$ satisfying $c(w) \ne (i, j)$ for all precolored neighbors w of v. The node v then selects $c(v) \leftarrow (dc(v), j)$.

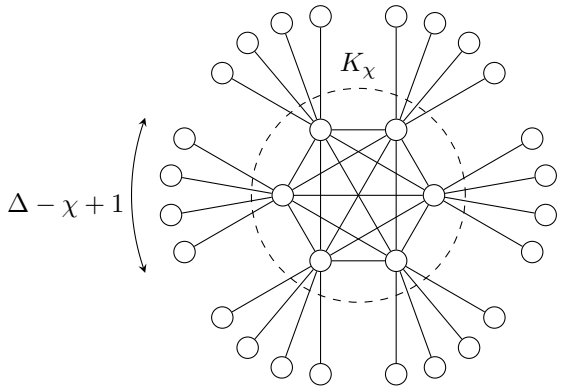


Figure 2 Graph whose single round color completion assigns at least $\chi \cdot (\Delta - \chi + 1)$ different colors across all instances. In this example $\chi = 6, \Delta = 9$.

By this algorithm, the color (i, j) selected by v is different from the color of any precolored neighbor of v. Also, (i, j) cannot be the selected color of any non-precolored neighbor w of v. This is because the default color $dc(w) = i'$ of w satisfies $i' \neq i$, and therefore, the selected color $c(w)$ of w is of the form (i', k) for some k, which must differ from (i, j) at least on the first component. Thus, the coloring c is proper. ◀

▶ Remark 9. In the absence of any preprocessing, [22] showed that we require $\Omega(\log^* n)$ rounds to color the graph even if it is just a path. To complement this, [22] also provides an $O(\log^* n)$ round algorithm that colors the graphs with maximum degree Δ with $O(\Delta^2)$ colors. In the full paper [1] we show that the same algorithm can be adapted to CC with at most $\lceil 23\Delta^2 \log_2 n \rceil$ new colors.

A consequence of the above is that one can readily adapt existing solutions of graph coloring to color completion. For example the results of [23], [8] can be extended to CC, with the number of colors used replaced by χ_{new} and retaining the same round complexities.

We complement the result of Thm. 8 with the following lower bound, see Figure 2 for the graph.

▶ **Theorem 10.** *For every integer χ, Δ, there exists a graph G with chromatic number χ and maximum degree Δ such that for every single round deterministic distributed algorithm $\mathcal{A}$, the total number of colors used by $\mathcal{A}$ over all recurrent instances of* CC *is at least $\chi \cdot (\Delta - \chi + 2)$ even after an arbitrary preprocessing of G.*

The single round algorithm can be extended to multiple round algorithm providing a similar color-round tradeoff as that of Maus' algorithm [23]. See the full paper [1] for the algorithm and discussions.

3.2 CC with fewer than $\Delta + 1$ colors

We next discuss coloring algorithms based on a preprocesing stage which, in many cases, use fewer than $\Delta + 1$ colors.

Our main result is an algorithm that, for a graph G with chromatic number χ, uses preprocessing, and in the recurrent stage solves any instance of CC with at most χ new colors in χ rounds. The algorithm operates as follows.

Preprocessing. The preprocessing stage computes a proper-χ coloring of the graph G. This is stored implicitly, i.e., each node v stores a single color (a positive integer) $dc(v)$. We call this coloring the initial coloring of G.

Online algorithm. We call the algorithm the "priority recoloring" algorithm. The set of nodes with the same initial coloring form an independent set which implies that nodes belonging to this set may be colored independently. We use the standard greedy algorithm to simultaneously color nodes with the same initial color in a single round. The initial colors are only computed to partition the original set of nodes into χ independent sets.
The input of each recurrent instance is a subset S of the nodes that were precolored, i.e., each $v \in S$ has a precolor $c(v)$. For convenience, consider $c(v) = 0$ for all $v \notin S$. The required output is a color completion of the precoloring: each node $v \notin S$ outputs a color $c(v) \in \mathbb{N}$ such that the colors assigned to all vertices form a proper coloring of the graph G.

The online algorithm $\mathcal{A}$ operates as follows.

- For $r = 1, 2, \ldots \chi$ rounds, do
  - If $dc(v) = r$ and $c(v) = 0$, then $c(v) \leftarrow \min(\mathbb{N} \setminus \Gamma(v))$, where $\Gamma(v) = \{c(w) | (w, v) \in E(G)\}$

For a given instance of the problem, χ^*_{un} (respectively, χ^*_{new}, χ^*_{all}) is the smallest possible value of χ_{un} (resp., χ_{new}, χ_{all}) over all possible proper color completions of the precoloring, and $\chi^{\mathcal{A}}_{un}$ (respectively, $\chi^{\mathcal{A}}_{new}$, $\chi^{\mathcal{A}}_{all}$) is the value of χ_{un} (resp., χ_{new}, χ_{all}) in the solution computed by the priority algorithm.

▶ **Observation 11.** *For any coloring, $\chi_{all} = \chi_{pc} + \chi_{new}$. In particular, $\chi^*_{all} = \chi_{pc} + \chi^*_{new}$ and $\chi^{\mathcal{A}}_{all} = \chi_{pc} + \chi^{\mathcal{A}}_{new}$.*

▶ **Lemma 12.** $\chi^{\mathcal{A}}_{new} \leq \chi$.

Proof. For every integer $k \geq 1$, let $\mathbb{N}_k = \{1, \ldots, k\}$. Let $M = \max \mathcal{P}_{pc}$, and let $FREE = \mathbb{N}_{M+\chi} \setminus \mathcal{P}_{pc}$ be the set of free colors (not used in the precoloring) up to $M + \chi$. Note that the cardinality of the set $FREE$ is at least χ. Let $\hat{F} = \{f_1, \ldots, f_\chi\}$ consist of the smallest χ integers in the set $FREE$.

By induction on k from 1 to χ, one can verify that during iteration k of the algorithm, the colors the algorithm uses for the uncolored vertices of default color $dc(v) = k$ are taken from $\mathcal{P}_{pc} \cup \{f_1, \ldots, f_k\}$. Hence $\mathcal{P}^{\mathcal{A}}_{un} \subseteq \mathcal{P}_{pc} \cup \hat{F}$, implying that $\chi^{\mathcal{A}}_{new} \leq |\hat{F}| = \chi$. ◀

▶ **Theorem 13.** *Consider a graph G with chromatic number $\chi = \chi(G)$. $\mathsf{CC}_{all}(\chi + \chi^*_{all} - 1)$ and $\mathsf{CC}_{new}(\chi)$ can be solved in χ SUPPORTED rounds.*

In the full paper [1] we discuss how tight these bounds are and prove the following lower bounds.

▶ **Theorem 14.** *(Lower bound for $\chi^{\mathcal{A}}_{new}$). For every deterministic distributed algorithm $\mathcal{A}$, and every integer $D > 3$, that solves CC with the guarantee that $\chi^{\mathcal{A}}_{new} < \chi$, there exists a graph G with diameter D and an instance of CC for which $\mathcal{A}$ takes $\Omega(D)$ SUPPORTED rounds.*

▶ **Theorem 15.** *For every integer $\chi \geq 2$ and deterministic algorithm $\mathcal{A}$ that solves CC with the guarantee that $\chi^{\mathcal{A}}_{new} \leq \chi^*_{new} + 1$, there exists a graph G with chromatic number χ and an instance of CC for which $\mathcal{A}$ takes χ SUPPORTED rounds.*

4 Locally Checkable Labellings

Locally Checkable Labellings (LCL) were first proposed by Naor and Stockmeyer [26].

Formal definitions and some examples can be found in the full paper [1].

4.1 Subgraph LCL's without Input Labels on Paths

In this section we consider a subset of recurrent LCL's, named *subgraph LCL's without input labels*, which were studied by Foerster et al. [16]. In subgraph LCL's, the online instances ask for a valid labelling for some (edge induced) subgraph of the given graph G. This sub-class of LCL's are easier to solve, but already captures several classical problems, such as finding a dominating set, maximal matching, maximal independent set, (k, l)-ruling sets etc.

We consider subgraph LCL on a path P_n. Before getting to the solution, we first remark that one may consider without loss of generality only LCL's with radius 1. Given an LCL problem of radius r, one may construct another LCL with the same round complexity but with radius 1 at the cost of increasing the output label size and the set of rules.

From a prior work (Theorem 3 in Foerster et al. [16]), we may infer that if the round complexity of Π in the LOCAL model is $o(n)$, then it must be $O(1)$ in the SUPPORTED model. This result is non-constructive, i.e., it argues that given a $o(n)$ round distributed algorithm, one can transform it into an $O(1)$ round algorithm. Additionally, it does not help categorize LCL problems that are $\Theta(n)$ in the LOCAL model. Some LCL problems (such as 2-coloring) are $\Theta(n)$ in the LOCAL model, but clearly $O(1)$ in the SUPPORTED model. One can also construct LCL's that remain $\Theta(n)$ in the SUPPORTED model. Furthermore, the proof offers no insight about the additional amount of memory per node that is needed for the preprocessing stage. The following theorem addresses the above questions. Note that as done in prior work, we treat the size of the description of Π as constant in the round complexity (in particular, $|\Sigma_{out}|$ and $|\Sigma_{in}|$ are constants).

▶ **Theorem 16.** *Let Π be a subgraph LCL with $|\Sigma_{in}| = 1$ and let P_n be a path on n vertices, then*

- *The* SUPPORTED *round complexity of Π is either $O(1)$ or $\Omega(n)$*
- *There exists a round optimal* SUPPORTED *algorithm for Π that stores only $O(1)$ bits as the output of the preprocessing phase*
- *The optimal* SUPPORTED *algorithm can be efficiently (polynomial in size of description of Π) computed by a centralized algorithm.*

4.2 Recurrent LCL's on Paths

In this section we look at a broader class of recurrent *in-labeled LCL's*, namely, LCL's augmented with input labels, wherein the online instances specify different input labels. The set of rules $\mathcal{C}$ and the set of output labels Σ_{out} remain the same across instances. The only components of the input that vary across instances are the input labels Γ^{in}. Subgraph LCL's, studied in Sect. 4.1, can be represented in this framework by encoding the adjacent edges that are present in the input labels for each vertex, hence in-labeled LCL's are a generalisation of subgraph LCL's. Problems such as finding a Client Dominating Set, Color Completion, Maximal Matching and in general variants of classical local problems with PFO and / or PCS instances fall into this category.

We show that even for these instances, the optimal round complexity is either $O(1)$ or $\Theta(n)$, thus extending the distributed speed up theorem in Foerster et al. [16]. However, so far we were unable to find a characterization as obtained in the previous subsection. Therefore,

we are left with a couple of intriguing open questions. First, we do not know any bounds on the constant of the running time in terms of the size of the LCL Π, namely, $|\Sigma_{in}| + |\Sigma_{out}|$. Second, we do not know if it is possible to decide the online round complexity in polynomial time given the description of the LCL.

▶ **Theorem 17.** *Let Π be a recurrent LCL problem whose online instances differ only in the assignment of input labels, then either Π can be solved in $O(1)$ SUPPORTED rounds or Π requires $\Omega(n)$ SUPPORTED rounds. Furthermore, there exists an algorithm with asymptotically optimal online round complexity using only $O(1)$ bits as the output of the preprocessing phase.*

Our proof of the above theorem is almost the same as that of Theorem 6 in [11] (for the LOCAL model) and Theorem 3 in [16] (for the SUPPORTED model). Note that, for paths, the above theorem is stronger than Theorem 3 of [16], which only translates $o(n)$ time algorithms in LOCAL to $O(1)$ time algorithms in SUPPORTED, whereas our argument also translates $o(n)$ time algorithms in SUPPORTED to $O(1)$ time algorithms in SUPPORTED. Theorem 3 of [16] holds for any hereditary graph family (for e.g. graphs with maximum degree Δ).

5 Conclusion and Future Directions

We discussed three recurrent problems and the advantages that a preprocessing phase has on their distributed round complexities. Several open problems remain, in particular:

- Can we extend state of the art algorithms for client dominating set on bounded arboricity graphs or other families? For example, the combinatorial algorithms of Morgan et al. [25] extend readily to the client dominating set problem. Is it possible to improve the round complexity of these algorithms in the SUPPORTED model?
- Can we obtain algorithms with better round complexities for color completion? We can perform color completion with at most χ new colors within χ rounds. Is this optimal?
- Extend the complexity gap theorems for trees - It is known that the distributed round complexity of LCL problems is either $O(1), \Theta(\log^* n), \Theta(\log n)$ or $\Theta(n^{1/k})$ for some integer k. (See [10] and [18]). These methods do not seem to be readily extendable to rooted trees in the SUPPORTED model.

5.1 Maximal Matchings and Maximal Independent sets

Apart from these questions, there are also other local problems of interest for which the round complexities in the SUPPORTED model are yet to be known. For example, consider the sub-graph maximal matching and sub-graph maximal independent set problem. For these problems tight bounds are known in the LOCAL model (see [5]), however these do not translate easily to the SUPPORTED model. We offer some partial results towards this direction. Proofs and detailed discussions appear in the full paper.

▶ **Theorem 18.** *Sub-graph Maximal Matching on graphs of arboricity a can be computed in $O(a)$ SUPPORTED rounds.*

▶ **Theorem 19.** *If subgraph MM can be solved in $o(\Delta)$ SUPPORTED rounds for all bipartite graphs, then subgraph-MM can be solved in $o(\Delta)$ SUPPORTED rounds for all graphs.*

▶ **Theorem 20.** *Sub-graph MIS for a graph G with chromatic number χ can be solved in χ SUPPORTED rounds.*

References

1 Akanksha Agrawal, John Augustine, David Peleg, and Srikkanth Ramachandran. Local problems in the supported model. Technical Report 2212.14542, arXiv, 2022. Full version of this paper. doi:10.48550/arXiv.2212.14542.

2 Noga Alon, Jarosław Grytczuk, Mariusz Hałuszczak, and Oliver Riordan. Nonrepetitive colorings of graphs. *Random Structures & Algorithms*, 21(3-4):336–346, 2002. doi:10.1002/RSA.10057.

3 Saeed Akhoondian Amiri, Stefan Schmid, and Sebastian Siebertz. Distributed dominating set approximations beyond planar graphs. *ACM Trans. Algorithms*, 15(3), jun 2019. doi:10.1145/3326170.

4 Brenda S Baker. Approximation algorithms for np-complete problems on planar graphs. *Journal of the ACM (JACM)*, 41(1):153–180, 1994. doi:10.1145/174644.174650.

5 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):1–30, 2021. doi:10.1145/3461458.

6 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, Jan Studenỳ, Jukka Suomela, and Aleksandr Tereshchenko. Locally checkable problems in rooted trees. In *Proc. 2021 ACM Symp. on Principles of Distributed Computing*, pages 263–272, 2021. doi:10.1145/3465084.3467934.

7 Leonid Barenboim. Deterministic (δ+ 1)-coloring in sublinear (in δ) time in static, dynamic, and faulty networks. *J. ACM*, 63(5):1–22, 2016. doi:10.1145/2979675.

8 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 *Proc. ACM Symp. on Principles of Distributed Computing (PODC)*, pages 437–446, 2018.

9 Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R. J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemyslaw Uznanski. LCL problems on grids. 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 101–110. ACM, 2017. doi:10.1145/3087801.3087833.

10 Yi-Jun Chang. The complexity landscape of distributed locally checkable problems on trees. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.DISC.2020.18.

11 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the local model. *SIAM J. Computing*, 48(1):122–143, 2019. doi:10.1137/17M1117537.

12 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the local model. *SIAM J. Computing*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

13 Yi-Jun Chang, Jan Studený, and Jukka Suomela. Distributed graph problems through an automata-theoretic lens. 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 31–49. Springer, 2021. doi:10.1007/978-3-030-79527-6_3.

14 Andrzej Czygrinow, Michal Hańćkowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In Gadi Taubenfeld, editor, *Distributed Computing*, pages 78–92, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-87779-0_6.

15 Vida Dujmović , Louis Esperet, Gwenaël Joret, Bartosz Walczak, and David Wood. Planar graphs have bounded nonrepetitive chromatic number. *Advances in Combinatorics*, mar 2020. doi:10.19086/aic.12100.

16 Klaus-Tycho Foerster, Juho Hirvonen, Stefan Schmid, and Jukka Suomela. On the power of preprocessing in decentralized network optimization. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1450–1458. IEEE, 2019. doi:10.1109/INFOCOM.2019.8737382.

17 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *2016 IEEE 57th Symp. on foundations of computer science (FOCS)*, pages 625–634. IEEE, 2016. doi:10.1109/FOCS.2016.73.

18 Christoph Grunau, Václav Rozhon, and Sebastian Brandt. The landscape of distributed complexities on trees and beyond. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 37–47. ACM, 2022. doi:10.1145/3519270.3538452.

19 Chetan Gupta, Juho Hirvonen, Janne H. Korhonen, Jan Studený, and Jukka Suomela. Sparse matrix multiplication in the low-bandwidth model. In *SPAA '22: 34th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 435–444, 2022. doi:10.1145/3490148.3538575.

20 Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1166–1179, 2021. doi:10.1145/3406325.3451081.

21 Ozan Heydt, Sebastian Siebertz, and Alexandre Vigny. Local planar domination revisited. In Merav Parter, editor, *Structural Information and Communication Complexity - 29th International Colloquium, SIROCCO 2022, Paderborn, Germany, June 27-29, 2022, Proceedings*, volume 13298 of *Lecture Notes in Computer Science*, pages 154–173. Springer, 2022. doi:10.1007/978-3-031-09993-9_9.

22 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

23 Yannic Maus. Distributed graph coloring made easy. In *Proc. 33rd ACM Symp. on Parallelism in Algorithms and Architectures*, pages 362–372, 2021. doi:10.1145/3409964.3461804.

24 Yannic Maus and Tigran Tonoyan. Local conflict coloring revisited: Linial for lists. *arXiv preprint*, 2020. arXiv:2007.15251.

25 Adir Morgan, Shay Solomon, and Nicole Wein. Algorithms for the minimum dominating set problem in bounded arboricity graphs: Simpler, faster, and combinatorial. 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 33:1–33:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.33.

26 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

27 Stefan Schmid and Jukka Suomela. Exploiting locality in distributed sdn control. In *Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 121–126, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491185.2491198.

A Holistic Approach for Trustworthy Distributed Systems with WebAssembly and TEEs

Jämes Ménétrey
University of Neuchâtel, Switzerland

Peterson Yuhala
University of Neuchâtel, Switzerland

Pascal Felber
University of Neuchâtel, Switzerland

Valerio Schiavoni
University of Neuchâtel, Switzerland

Aeneas Grüter
University of Bern, Switzerland

Julius Oeftiger
University of Bern, Switzerland

Marcelo Pasin
University of Neuchâtel, Switzerland

Abstract

Publish/subscribe systems play a key role in enabling communication between numerous devices in distributed and large-scale architectures. While widely adopted, securing such systems often trades portability for additional integrity and attestation guarantees. Trusted Execution Environments (TEEs) offer a potential solution with enclaves to enhance security and trust. However, application development for TEEs is complex, and many existing solutions are tied to specific TEE architectures, limiting adaptability. Current communication protocols also inadequately manage attestation proofs or expose essential attestation information. This paper introduces a novel approach using WebAssembly to address these issues, a key enabling technology nowadays capturing academia and industry attention. We present the design of a portable and fully attested publish/subscribe middleware system as a holistic approach for trustworthy and distributed communication between various systems. Based on this proposal, we have implemented and evaluated in-depth a fully-fledged publish/subscribe broker running within Intel SGX, compiled in WebAssembly, and built on top of industry-battled frameworks and standards, i.e., MQTT and TLS protocols. Our extended TLS protocol preserves the privacy of attestation information, among other benefits. Our experimental results showcase most overheads, revealing a 1.55× decrease in message throughput when using a trusted broker. We open-source the contributions of this work to the research community to facilitate experimental reproducibility.

2012 ACM Subject Classification Security and privacy; Security and privacy → Distributed systems security; Security and privacy → Trusted computing; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Publish/Subscribe, WebAssembly, Attestation, TLS, Trusted Execution Environment, Cloud-Edge Continuum

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.23

Supplementary Material *Software*: `https://github.com/JamesMenetrey/unine-opodis2023` archived at `swh:1:dir:8a6b3762b7da58b4af2de849dc93bca9f69aaf67`

Funding This publication incorporates results from the VEDLIoT project, which received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957197.

1 Introduction

Publish/subscribe (pub/sub) systems [18] have become foundational for seamless intercommunication of a wide range of devices, from IoT ecosystems to large-scale cloud-based services, i.e. the cloud-edge continuum [65]. These systems enable efficient and scalable data

distribution among distributed entities by decoupling data producers from data consumers. Given their widespread adoption, notably in cloud computing [3, 24, 43], several pub/sub systems have been proposed with the clear goal of providing additional privacy guarantees [50]. However, the nodes participating in these systems are implicitly trusted, relegating security concerns primarily to the protection of communication channels or leveraging heavyweight cryptographic primitives [31]. This limited security approach leaves data on the processing components vulnerable to potential threats, especially in decentralised and heterogeneous environments. Notably, high-privileged actors within the pub/sub nodes, i.e., operating system or hypervisor, may compromise the confidentiality and integrity of data. Similarly, they could leak critical cryptographic material, e.g., private keys of certificates. Leaking certificate keys is especially concerning as these keys serve as the foundation for the authentication process among pub/sub participants, thereby putting user privacy and data integrity at stake.

In both the consumer market and cloud providers, trusted execution environments (TEEs) present a solution to strengthen the integrity and confidentiality of data in use, especially on nodes that may not be inherently trustworthy. TEEs provide *enclaves*, e.g., hardware-protected memory regions, where sensitive computations are completely isolated from other software executed on the same platform. Such secure enclaves significantly elevate the security posture of systems like pub/sub, safeguarding not just the communication but also the processing and data on such nodes from malicious actors. As a keystone feature of TEEs, attestation enables a remote entity to verify the authenticity, configuration, and state of a trusted environment using cryptographic proofs, ensuring the enclave runs the intended software without being tampered with or compromised [40]. In pub/sub systems deployed over untrusted infrastructures, attestation ensures data and its processing within the TEE enclave are kept confidential and untampered.

Nonetheless, developing applications for TEEs is challenging due to their specific programming paradigms and SDKs, requiring massive efforts when writing or porting existing software [25]. In addition, while many pub/sub systems exploit TEEs to protect data in use (covered in §3.1), these usually tie closely to a specific TEE architecture, which is limiting in the heterogeneous environments (e.g. using different CPUs) considered in this work. Beyond this, current secure communication protocols fail to transport attestation proofs, do not maintain the privacy of these messages when supported, or cannot expose X.509 certificates issued by global authorities (covered in §3.2). This leads to the use of ad-hoc implementations, which fall short in offering a consistent solution across the cloud-edge continuum.

In this paper, we solve the major challenges associated to writing secure and portable pub/sub systems by relying on WebAssembly (Wasm), an open-standard binary instruction format. Wasm's architecture-neutral design abstracts the complexity of hardware and TEE requirements, making it particularly suitable as a compilation target for pub/sub systems in heterogeneous hardware environments, such as the cloud-edge continuum. We further protect the communication channels by extending the industry standard TLS protocol for embedding attestation evidence in the handshake while maintaining compatibility with the original specifications. This ensures the authenticity of the executing code of parties within the pub/sub system, thereby mitigating the risk of malicious entities impersonating or modifying these components. Additionally, certificate keys are safeguarded within the TEEs, preventing potential leaks. Besides, we developed a proof-of-concept that encapsulates the full implementation of a standard pub/sub broker and a TLS library, enabling the termination of TLS channels directly in the TEE, which has been proven complex in prior work [8]. We achieve this by using Wasm as a compilation target for both software, limiting the number of code changes required to make them compile and run within TEEs. While our prototype is focused on cloud environments by leveraging Intel SGX, we outline the

trusted primitives required for our proposal to be compatible with other platforms, including edge and IoT devices. We summarise our contributions as follows: *(1)* a unified strategy that secures a standard pub/sub system using TEEs with attestation for security, and compiled in Wasm for seamless cloud-edge communication while minimising code changes, *(2)* an extension to the TLS communication protocol, facilitating the confidential exchange of attestation evidence, thereby affirming the authenticity of actors within the pub/sub system and *(3)* an open-source implementation of a pub/sub broker using Intel SGX for cloud environments, with a suite of benchmarks aimed at evaluating the impacts of Wasm, the TEE and attestation, in comparison with state-of-the-art work. Our evaluation reveals that our system delivers messages at 1.55× slower than baseline throughput, yet provides portability and the robust security guarantees of TEEs.

2 Background

2.1 Pub/sub systems

A publish/subscribe system (often called pub/sub) is an asynchronous architecture for message passing that connects two different types of entities: publishers and subscribers. Publishers send messages (or events), being unaware of who is interested in receiving such messages. Messages sent are usually marked with an arbitrary type or a category. Subscribers express their interest in specific types of messages and receive them when they are published. Pub/sub is commonly used in event-driven distributed systems such as the Internet-of-things, since they simplify the communication between different entities.

Pub/sub messages are usually passed from publishers to subscribers using an intermediary broker. Brokers receive messages from publishers and forward them to subscribers who have expressed interest in the corresponding topics. Message brokers are responsible for routing and delivering messages to the appropriate subscribers. Examples of message brokers include Apache Kafka, RabbitMQ, MQTT brokers, and cloud-based pub/sub services like Amazon SNS (Simple Notification Service) or Google Cloud Pub/Sub.

One of the key advantages of pub/sub is decoupling since publishers and subscribers do not need to know each other. Pub/sub systems can also be designed to be highly scalable and fault-tolerant by using replicated, interconnected brokers. This flexible architecture allows components to be added or removed without affecting the system.

Security in pub/sub systems is dealt with when publishers and subscribers connect to their brokers. Brokers then usually implement authentication and access control before publishing or subscribing to messages. They can also establish encrypted connections (e.g. TLS) to implement message privacy. By implementing attestation, we offer a stronger guarantee for all the components. Publishers, subscribers, and brokers are guaranteed to be who they say they are, and all can ensure that their counterparts execute appropriate (correct) software.

2.2 Trusted execution environments

Trusted execution environments (TEEs) offer secure and isolated areas of a computer system. TEEs shield sensitive data and code from unauthorised access, compromised libraries or operating systems. This isolation is hardware-enforced, considerably minimising the software-based vulnerabilities. In contrast to trusted platform modules (TPMs) [1], which are limited to a specific and finite set of operations, TEEs enable the execution of arbitrary code, providing greater flexibility and versatility. Code executed inside a TEE can communicate with untrusted programs or libraries, but such interactions are tightly controlled. TEE-specific mechanisms ensure data is secure during transfers into and out of an enclave.

We focus on Intel Software Guard Extensions (SGX) [15, 47], a popular and widely used TEE architecture. Available in server-grade CPUs, Intel SGX manages and runs code within secure enclaves, which interact with the other processes and operating system by transitioning the enclave boundary using specific instructions, i.e., `ECALLs` and `OCALLs`. Enclaves use `ECALLs` to receive function calls from untrusted applications outside the enclave and `OCALLs` to invoke functions outside from within the enclave. The enclave's memory is encrypted by a dedicated MMU, ensuring protection against attackers with access to memory data.

2.3 Attestation

Attestation is a security mechanism enabling an *attester* to prove its identity and integrity to an *verifier*. In the realm of TEEs, attestation allows a TEE to prove its configuration, identity, and state to another device or service. Most TEE implementations support attestation, either built-in (as in SGX [32] and its successor TDX [59, 14], AMD SEV-SNP [4]) or by means of research prototypes [42]. The primary objective is to ensure the attester is genuine, unmodified, and trustworthy. To achieve this objective, attestation leverages proofs (i.e. *evidence*), a cryptographically signed document composed of *claims*, i.e., pieces of asserted information, like the hash of a code running in the TEE, i.e., *measurement*. The verifier is bootstrapped with *reference values*, compared against the received claims for validation.

In Intel SGX, attestation is paramount for establishing the authenticity and integrity of code and data inside an enclave. Intel controls a set of keys, intrinsically linked to the hardware and identity of the enclave. Such keys are inaccessible from the enclaves themselves, and are used by the processor and Intel signed software to issue trustworthy evidence. Intel is considered trusted, as they provision a subset of keys into the hardware. Within this assumption, third parties can trust evidence produced by the TEE.

2.4 Communication channel and attestation binding

Communication channel. Over the past decades, several standards have emerged to establish trustworthy communication between two entities. Among them, Transport Layer Security (TLS) stands out as the most prominent protocol for such tasks. This cryptographic protocol ensures confidentiality, integrity, and authentication for secure communication. A typical TLS session begins with a phase known as the *handshake*. This TLS handshake acts as a negotiation process, where the two parties decide on encryption settings and authenticate one another before exchanging secure data. More precisely, the latest version of the TLS handshake (1.3 at the time of this writing) is composed of *1)* the *ClientHello* sent by the client, notably containing the supporting cipher suites, key agreement and anti-replay mechanisms, *2)* the server sends the *ServerHello* response, detailing analogous parameters, while signifying the handshake's conclusion, and finally *3)* the handshake wraps up as the client reciprocates with a similar acknowledgement, accompanied by data from a higher-level protocol, that is wrapped within TLS, e.g., HTTPS. Additionally, during steps 2 and 3, both entities can opt to present an X.509 certificate. These serve as identity validators, embedding trust through a chain of digital signatures rooted in trusted Certificate Authorities (CAs). Building on this, many existing attestation protocols that bind with TLS typically augment the handshake, the certificate, or a combination of the two, facilitating the negotiation of security parameters and the exchange of attestation evidence.

Channel binding. Channel binding [7] ensures that an entity participating in a secure communication, typically via protocols such as TLS, is indeed the entity that has undergone attestation. It establishes that the communications are with the attested environment,

preventing possible relay attacks where a malicious party might relay attestation challenges to a genuine system and then claim the legitimate evidence as their own. To secure trusted communication, attestation evidence is typically integrated into the early phases of the communication protocol. However, combining attestation with these protocols introduces challenges, including increased latency and additional trade-offs like binding the system to a particular TEE technology.

As later outlined in this paper, the proposed solution refines the TLS protocol with minimal enhancements. Unlike traditional approaches that embed attestation in custom fields in the broker's X.509 certificates, our method leverages a custom TLS 1.3 encrypted extension. This approach reduces the need for additional communication roundtrips between the client and server. It preserves the broker's ability to use certificates issued by recognised CAs, an aspect overlooked in previous studies.

2.5 WebAssembly

WebAssembly (Wasm) [27] is an open standard that provides a portable and executable bytecode, tailored for running applications at near-native speed. While its initial design targeted web applications written in traditional native languages like C/C++, Rust, and Go, its remarkable performance has led to its adoption not just within browsers but also in standalone environments [27, 74]. Wasm was developed as a cross-platform, language-agnostic compilation target, optimised for easy integration in resource-limited environments, making it a prime candidate for TEE development. Notably, several specialised runtimes, such as Twine [41, 17] for Intel SGX and WaTZ [42] for Arm TrustZone, have been developed to leverage Wasm's capabilities within state-of-the-art TEE implementations, and natively support key TEE primitives such as attestation. For Wasm applications, particularly in standalone settings, the WebAssembly System Interface (WASI) [44] provides direct access to essential OS services, traditionally managed by POSIX and system calls. WASI facilitates seamless communication with a handful of system components, ranging from file systems and networking to time management and random number generation, regardless of the underlying platform. Runtimes optimised for TEEs map these system calls to trusted services, typically offered by the TEE manufacturers, ensuring that sensitive data remains secure and protected within the enclave.

We leverage Wasm as a versatile, cross-platform environment to model our pub/sub proposal seamlessly across the cloud-edge continuum and within TEEs as the execution context. This approach offers unmatched portability across many processor and TEE architectures. When combined with TEEs, Wasm hardens our solution against powerful threats, such as the operating system, the hypervisor or malicious system administrators. This security guarantee is especially important in edge computing scenarios where the local infrastructure typically lacks physical security measures as established in cloud environments.

3 Related Work

3.1 Pub/sub with TEEs

In the dynamic world of distributed systems, pub/sub mechanisms have consistently gained traction, acting as the core foundation for a variety of applications and architectures. Many approaches have been conducted in research to bring dependability and safety regarding pub/sub systems, and in many different directions [72]. An explored aspect is encryption schemes of data transferred through the brokers, preserving the privacy of the communicated

Table 1 Comparison of the state-of-the-art pub/sub systems shielded by TEEs. ●, ◑, ○ mean fully, partially and not implemented, respectively. ? denotes not disclosed details. Features description: *1) Communication protocol*: protocol used by peers to interact with the broker. *2) Fully enclaved broker*: broker operates within the TEE instead of only securing specific components within the secure environment. *3) Peer authentication*: broker authenticates the peers while establishing communication. *4) Peer attestation*: broker attests the peers while establishing communication. *5) Broker authentication*: peers authenticate the broker while establishing communication. *6) Broker attestation*: peers attest the broker while establishing communication; ◑ means that only publishers are attested. *7) Persistence of messages*: system's capability to store messages for future delivery (e.g., when subscribers might be temporarily offline). *8) Idiomatic pub/sub architecture*: system adheres to the principles of the pub/sub paradigm. *9) Open source*: implemented solution is freely available to the public via an open-source repository. *10) TEE technology*: denotes the TEE used by the proposed system, or its capacity to operate agnostically across various TEEs.

	[53] SCBR	[6] PubSub-SGX	[73] MagikCube	[61] MQT-TZ	[52] Pei et al.	This work
Comm. protocol	TLS	TLS	TLS	TLS	?	TLS
Fully enclaved broker	●	●	●	○	○	●
Peer authentication	○	●	●	●	?	●
Peer attestation	○	○	○	○	?	●
Broker authentication	●	●	●	●	?	●
Broker attestation	◑	●	●	○	?	●
Persistence of messages	○	●	○	○	○	●
Idiomatic pub/sub arch.	○	●	●	●	●	●
Open source	○	○	○	●	○	●
TEE technology	SGX	SGX	SGX	TZ	SGX	*Agnostic*

TEEs references: SGX (Intel SGX), SNP (AMD SEV-SNP), TDX (Intel TDX), TZ (Arm TrustZone)

information [29, 46, 10, 12, 38, 21, 11]. Cryptographic-based privacy protection schemes focus on encrypting events and subscriptions, and then performing ciphertext matching between them. However, they often suffer from scalability issues as matching time complexity grows with the number of subscriptions, leading to diminishing performance.

More recently, researchers have investigated the potential of Intel SGX as a secure environment for confidential data processing. Leveraging TEEs has shown potential for enhanced performance over cryptography-based methods, as highlighted by SCBR [53]. Their study details a custom content-based routing engine operating within an SGX enclave. PubSub-SGX [6] introduced a scalable approach, using a load balancer that manages multiple matchers, each operating within individual enclaves. Following this, MagikCube [73] added an authentication service to the pub/sub system, thereby enhancing broker trust among publishers and subscribers using SGX. Finally, Pei et al. [52] further refined this paradigm by optimising subscription matching times using cryptographic methods, with SGX facilitating comparison tasks but does not disclose how secrets are exchanged. In contrast with previous work, our proposal prioritises establishing an initial trust across all pub/sub participants by mutual attestation. We do it by encapsulating conventional pub/sub systems within TEEs, ensuring genuine execution environments and trustworthy implementations.

Other prior studies have chosen a different strategy using TrustZone, Arm's TEE. In this setup, the device is divided into the *normal world* (the standard OS) and the *trusted world* (the TEE). MQT-TZ [61] migrated the broker's data management component within this trusted world. Publishers and subscribers negotiate symmetric keys with the broker,

which are generated inside the TEE. This approach ensures that data is encrypted during transmission. Conversely, our proposal enhances the threat model of MQT-TZ by relying on entity attestation in the pub/sub system and hosting the TLS endpoint directly within the TEE, avoiding handling cryptographic materials outside the TLS protocol.

Comparison. Table 1 offers a comprehensive summary of state-of-the-art proposals for securing pub/sub systems with TEEs, focusing on features relevant to our study.

3.2 Binding Communication Protocols and Attestation using TEEs

Research has extensively explored the integration of secure communication protocols with attestation. While our focus primarily lies on solutions leveraging TEEs, we also recognise the significant contributions from prior work that leveraged TPMs as trust anchors [23, 64, 22, 63, 5, 78, 9, 51, 70, 69]. Readers can refer to [33, 48] for a comprehensive review of these works. Our analysis omits explicitly works that discuss attestation through TEEs but do not bind attestation evidence to a communication channel [39, 16, 20, 76].

Intel proposed a remote attestation protocol for key exchange based on SIGMA [34, 28], binding SGX enclaves with communication channels. Subsequent solutions aimed to establish trusted communication channels with enclaves using custom message exchanges [62, 35, 42].

Table 2 Comparison of the state-of-the-art channel binding solutions. ●, ○ mean fully and not implemented, respectively. ? denotes not disclosed details. "—" denotes not applicable comparisons. Features description: *1) Baseline protocol*: protocol upon which the proposal is built. *2) No change in TLS specification*: proposal respects the TLS specifications, ensuring compatibility with pre-existing TLS implementations. *3) Attestation privacy*: protocol maintains confidentiality of attestation evidence. *4) Mutual attestation*: communicating entities engage in a mutual attestation process. *5) Evidence per session*: each communication session is uniquely associated with specific attestation evidence. *6) Attestation Privacy*: all attestation-related messages are encrypted; ◑ means that only a portion of the messages remains confidential. *7) TEE-agnostic*: independent of any specific programming language or TEE-specific SDK; ◑ means theoretical approach proposed, but no agnostic implementation. *8) Endpoint in enclave*: endpoint application fully resides within the TEE. *9) Support global CAs*: broker-displayed certificates can be vouched for by globally recognised CAs. *10) Open source*: implemented solution is freely available to the public via an open-source repository. *11) TEE technology*: denotes the TEE that the attestation API currently supports, or its capacity to operate agnostically across various TEEs.

	[28] SGX EPID	[62] Shepherd et al.	[42] WaTZ	[33] RA-TLS	[26] Palæmon	[48] TSL	This work
Baseline protocol	Custom	Custom	Custom	TLS	TLS	TLS	TLS
No change in TLS spec.	—	—	—	●	●	●	●
Attestation privacy	○	●	○	○	●	●	●
Mutual attestation	○	●	○	●	○	●	●
Evidence per session	●	●	●	○	○	●	●
Endpoint in enclave	●	●	●	●	●	○	●
Attestation privacy	○	◑	○	◑	?	◑	●
TEE-agnostic	○	◑	◑	○	○	◑	●
Support global CAs	—	—	—	○	○	○	●
Open-source	●	○	●	●	○	○	●
TEE technology	SGX	TZ	TZ	SGX	SGX	*Agnostic*	*Agnostic*

TEEs references: SGX (Intel SGX), SNP (AMD SEV-SNP), TDX (Intel TDX), TZ (Arm TrustZone)

While these initial efforts in remote attestation provided valuable insights, their custom nature makes them challenging to integrate into existing software. In contrast, our approach harnesses TLS, the leading industry standard for secure communication.

More recently, further research [66, 26, 48, 33, 49] have suggested using TLS for communication and modifying the X.509 certificates in order to include additional fields related to attestation. Although this method takes advantage of the protocol's standardisation to address the earlier concern, it ties the attestation mechanism directly to the certificates exposed by the endpoints. This direct connection implies that certificates must be dynamically generated, which restricts them from being signed by global CAs like *Let's Encrypt*, commonly used for domain validation certificates. Opting for a different route, we used the encrypted extensions of the TLS protocol for carrying evidence of the server. Consequently, our approach is compatible with certificates endorsed by global CAs. This is particularly beneficial if the endpoint owns a separate network-level identity, like a DNS or domain name.

Comparison. Table 2 offers a comprehensive summary of cutting-edge research dedicated to binding communication channels with attestation, focusing on features relevant to our study.

4 Design Considerations

We explain the threat model of our design, highlight its security goals and cover the trusted primitives that must be available on a system to support our proposal.

4.1 Threat Model

Our approach relies on a few key trusted components, which are essential for our system to be deemed trustworthy. We further discuss these elements in the remainder of this section.

TEEs. Our proposal leverages the protection offered by TEEs for securing the execution of applications and enforcing strong isolation against powerful attackers, such as the OS or the hypervisor. Given that our proposal is TEE-agnostic, we highlight the minimal requirements to uphold trust in the pub/sub system and attestation mechanism. We assume the application code can be inspected but cannot be subverted. Data in use remains confidential and cannot be read unless granted by the TEE. The hardware and software strictly required to run a TEE instance are considered trusted. Furthermore, the TEE offers a remote attestation mechanism backed by genuine entities responsible for validating the trustworthiness of attestation evidence. Although we do not address side-channel or denial-of-service attacks [13, 77, 19], there exist measures for these in various TEE designs [2].

OS. The OS follows an *honest-but-curious* threat model, posing no threat to the trusted environment but interested in gathering sensitive information. Consequently, it can monitor all communication within the pub/sub system. Assuming the OS starts to behave maliciously, the trusted computing base (TCB) remains confidential and doesn't malfunction, though it might become unresponsive. The applications running in the TEE are carefully developed to ignore abnormal responses and abandon execution in such cases.

Wasm. We presume that the Wasm runtime is implemented correctly and does not contain vulnerabilities. The Wasm runtime acts as a shim library by encapsulating Wasm applications and uses trusted APIs from the TEE SDK for system interactions or sanitises the interaction with the untrusted OS when no secure option is available. While side-channel attacks might target Wasm and TEEs [54], they fall beyond the scope of this study.

Cryptography. As we embed a TLS library and enhance it to integrate attestation concerns, we suppose the correct implementation of the cryptographic ciphers and operations. Additionally, we also presume that standard cryptographic techniques cannot be subverted and are hardened against side-channel attacks.

4.2 Security requirements

We propose a series of requirements for establishing trusted communication channels between the actors of pub/sub systems:

(SR1): Support global CAs certificates: brokers shall support exposing certificates issued by globally recognised CAs.

(SR2): Trust assurance: pub/sub actors shall attest the TCB of the participant they connect with and must be similarly verified in return.

(SR3): Channel and attestation bindings: communication channels must be linked to newly created attestation evidence, preventing replay or collusion attacks [48].

(SR4): Attestation privacy: attestation information shall remain confidential between the endpoints of a given communication channel.

(SR5): Pub/sub privacy: all the data bound to the pub/sub system shall be inaccessible to outside actors, including the OS, kernel and hypervisor of the broker and peers.

(SR6): Pub/sub narrow scope: the peers shall only publish or subscribe to the topics as required for their needs.

4.3 Trusted primitives

We identified a core set of trusted primitives that any system must support to host our proposal securely. These primitives are provided by the secure environment or the Wasm runtime:

- **Isolated execution context**: this ensures the runtime remains secure and unaffected by any other applications running concurrently on the system. All major TEE manufacturers support at least one Wasm runtime, though the isolation paradigm and threat model vary.
- **Attestation capabilities**: the TEE must expose two primitives to the Wasm runtimes: *generate* and *verify*. The former primitive generates evidence for proving the trustworthiness of the secure environment with additional data attached, such as a nonce and a public key, while the latter confirms the validity of this proof.
- **Network communication**: the Wasm runtimes require access to a network API that handles socket operations and the transfer of data.
- **File system access**: if the pub/sub broker needs to store publications for future delivery, the system should offer secure ways to save and access these files.

Other concerns, such as encryption and pub/sub protocol logic, are platform-independent and addressed using dedicated software compiled into Wasm.

5 Architecture

In this section, we outline the architecture and design rationale of our proposal. First, we explain the changes made to the TLS protocol. Next, we discuss the security enhancements of our pub/sub system. We wrap up with insights into specific implementation details.

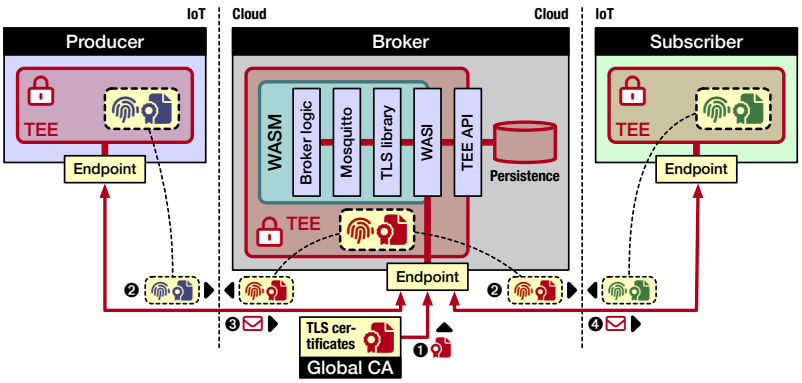


Figure 1 The overall architecture of our proposal. (,) mean X.509 certificate and attestation evidence, respectively. The colours of these icons correspond to the actor owning them.

5.1 Overview

We designed our proposal as a versatile system capable of running on numerous processor architectures. The system isolates security-sensitive pub/sub operations of peers and brokers, ensuring the authenticity of connecting machines using TEEs and mutual attestation. A key aspect of our design is the adoption of Wasm, facilitating cross-platform support across various TEE architectures. More specifically, we rely on trusted Wasm runtimes [41, 42], to host a secure pub/sub system with its associated dependencies such as TLS libraries, thus covering the large spectrum of the cloud-edge continuum.

Figure 1 illustrates the key components and entities within our pub/sub solution. Serving as the central hub, the broker first acquires a certificate from a global CA for its TLS endpoint (❶). Peers then initiate secure communication with the broker via our enhanced TLS handshake, performing mutual remote attestation by exchanging their X.509 certificate for authentication and attestation evidence (detailed in Section 5.2). This ensures that the broker and peers are trustworthy (❷). The publisher generates data and transmits it to the broker's TEE (❸). The broker, in turn, relays this data to the subscriber's TEE (❹). Both the publisher and the subscriber use the same technology stack in their respective TEEs, although this detail is omitted in the figure for clarity.

5.2 Attesting communication channels

We enhanced the TLS protocol to integrate the exchange of attestation evidence when a peer is communicating with the broker. This exchange of information occurs in the TLS handshake, as depicted in Figure 2. Our enhancements are highlighted in bold.

Attestation protocol. The first message of the handshake (①) is sent by the peer, which is comprised of an encrypted attestation request (**AttReq**), indicating that the peer is establishing a TLS channel requiring attestation. Contrary to prior work, our protocol does not specify TEE architectures in this message, as our solution supports the verification of all the types of TEEs that can be used in the broker. Similarly, our protocol is the only proposal that studies the encryption of the attestation request, as further explicated below.

When received, the broker answers with a message (②) composed of its evidence (**AttServer**), freshly generated and bound to the TLS session. It is worth noting that we deliberately chose not to embed attestation data within the broker's X.509 certificate so the endpoint can use certificates endorsed by globally recognised CAs, satisfying **(SR1)**.

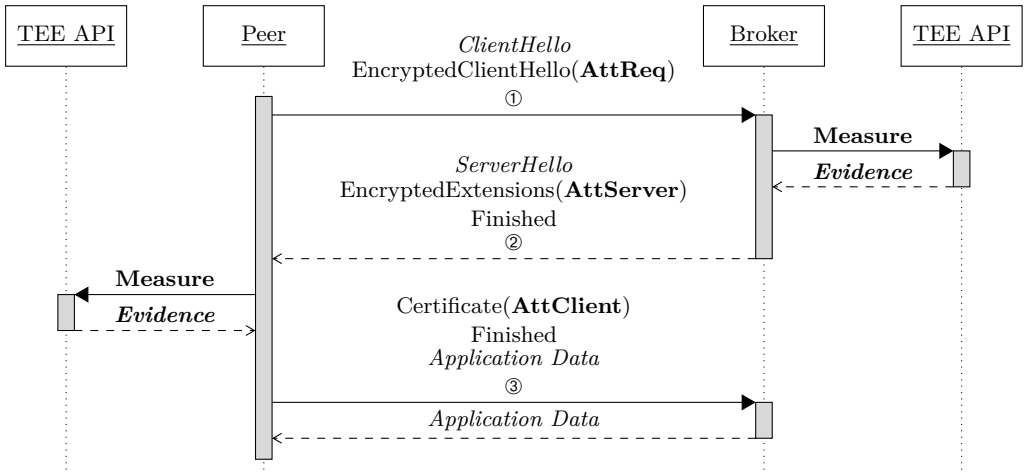


Figure 2 Enhanced TLS 1.3 handshake with attestation. New elements are mentioned in bold.

Subsequently, the peer verifies whether the evidence of the broker is genuine and checks if the evidence is bound to the current TLS session (to counter reuse attacks). Moreover, it compares the code measurement of the broker to a known reference value, indicating that the code of the broker is trusted. Since a global CA issues the broker's X.509 certificate, the peer verifies that the DNS or domain name of the broker matches the identifier exposed in the X.509 certificate [57]. If these conditions are fulfilled, the peer issues its evidence (**AttClient**) and sends it to the broker (③), also bound to the TLS session and embedded within the peer's X.509 certificate. We had to rely on a custom X.509 extension for the peer, as TLS 1.3 does not provide an extension point for the last message of the handshake.

Lastly, the broker also verifies whether the evidence of the peer is genuine, and checks if the evidence is bound to the current TLS session. Similarly, the broker ensures that the code measurement of the peer matches a known reference value, indicating that the code of the peer is trusted. Once completed, the peer and the broker mutually attest themselves, ensuring their respective TEE is trustworthy, satisfying **(SR2)**. As a result, the TLS handshake is ended, and the applications may start pub/sub communication.

Binding the handshake with attestation evidence. Integrating the attestation mechanism in the TLS handshake has many benefits: the execution time is optimised since no additional round-trips are necessary, and the TLS 1.3 standard is respected by using extension points as designed by the protocol. Besides, we strongly bind freshly generated evidence to individual TLS sessions, using unique TLS keying materials computable by both parties (RFC 5705) [55], satisfying **(SR3)**. This prevents replay and collusion attacks, which would affect published past evidence otherwise. Since our pub/sub system and TLS library are compiled in Wasm, evidence binding within TLS handshakes is portable across different TEE architectures.

Securing the attestation information. We use three distinct encryption mechanisms to ensure the confidentiality of the attestation information, to comply with **(SR4)**. First, we opted to use the recently-introduced *encrypted client hello* (ECH) for TLS [56], which is currently an IETF draft to communicate early information while preserving its privacy in the *ClientHello* message. To the best of our knowledge, we are the first to leverage this draft to protect the attestation request (**AttReq**) in the TLS handshake. In a nutshell, the *ClientHello* message is split into two parts: the outer message, which is in plain text, and

the inner message, which is encrypted using a public key, typically distributed using DNS infrastructure. Second, we rely on the TLS 1.3 encrypted extension in the broker reply (i.e., the `ServerHello` message), so the broker's attestation evidence remains confidential. This has been made possible because the two parties can derive a shared secret for encryption up to this point. Finally, the peer's certificate, which contains the peer's evidence, is also protected, since the entire third message of the handhake is encrypted by design.

5.3 Securing pub/sub systems

We leverage TEEs and trusted Wasm runtimes for implementing our pub/sub design, ensuring strong hardware isolation of both code and data. This design remains versatile, working with various TEE architectures as long as they offer the trusted primitives for the Wasm runtimes (as in §4.3). When paired with the mutually attested TLS protocol, we shield data in use and in transit, satisfying **(SR5)**. As a pub/sub software, we selected Mosquitto [37], a well-known and open-source message broker that implements the latest version of the MQTT protocol. We have chosen Mosquitto because it is lightweight and suitable for use on all devices, from low-power IoT devices to cloud servers. Besides, we used WolfSSL, an embeddable cryptographic library, enabling the host of the TLS termination directly in the TEE, so external systems cannot eavesdrop on the communication, nor alter the code that maintains the endpoint. We needed to compile Mosquitto in Wasm, which required slight modifications to the source code. Regarding WolfSSL, we reused one of the extension works of a trusted runtime [41], which already compiled this library in Wasm.

System interactions. Mosquitto, like most software, is required to perform system calls for interactions with the outside world. Typically, this is the case for the socket API when exposing the TLS endpoint. For such usages, we rely on WASI, which translates the system calls to the underlying OS seamlessly. As the private and session keys of TLS are located within the TEE, the communication remains confidential against eavesdropping attempts, even when system calls are monitored. Besides, Mosquitto can persist undelivered messages in a database for later transmission to offline subscribers. The WASI specification includes a file system API for data storage, while TEEs typically provide means to save files via a trusted API securely. In contrast to the socket API, Wasm runtimes use that TEE API to transparently encrypt files, ensuring the confidentiality of the stored messages.

Securing and narrowing the pub/sub data access. Our proposal ensures that brokers and peers are trustworthy as they are mutually attested. As such, we inherently restrict adversaries to propagate and observe messages going through the pub/sub system. Nonetheless, we propose to reduce further the scope of peers using access control lists (ACLs), which is a built-in functionality of Mosquitto. When paired with the X.509 certificates presented in the TLS handshake, this feature enables to authenticate peers without requiring additional usernames or passwords, as this is typically the case using Mosquitto. This narrowed publication and subscription scope satisfies **(SR6)**. Future work may further adapt the ACLs to be bound to evidence, provided that code measurements differ from each actor.

Use cases. Our approach is applicable to a variety of distributed pub/sub systems, which supports scalable communication by decoupling publishers from subscribers and ensures secure messaging through mutual attestation. This was demonstrated in our European-funded project in collaboration with Siemens for secure data processing at edge nodes and capturing results on an MQTT messaging bus, as well as with the Byzantine fault-tolerant (BFT)

system of the University of Lisbon for maintaining and ensuring trust in the attestation reference values [67]. To balance the need for compactness with computational capability, we used Intel NUCs as edge nodes, which are small, efficient PCs equipped with processors that support Intel SGX. This allows the publisher on the edge nodes to verify whether the broker is genuine based on evidence and the code hash contained in that payload. This code hash is then compared to the trusted reference values stored in the BFT-resilient system. The evaluation section (§6) details how the broker performs when handling heavy loads from multiple publishers and subscribers, as encountered in these distributed environments.

5.4 Implementation

We developed a prototype of a trusted pub/sub broker using Intel SGX to better understand the practical impacts and performance overhead associated with our proposal. As such, we used Twine [41], a trusted Wasm runtime designed to secure applications on Intel SGX. We target the latest Intel SGX technology, named *Intel Scalable*, for creating enclaves with an enclave page cache (EPC) up to 512 GiB, whereas prior work was limited to 128 MiB, which was a performance bottleneck when reaching that threshold.

Generating attestation evidence. We provided the two trusted primitives of attestation (as in §4.3), i.e. *generate* and *verify*, which use the Intel SGX attestation capabilities. The former primitive calls the underlying TEE API to create a payload in the JSON format, which is later communicated in the TLS handshake. That payload notably contains the type of TEE and the evidence itself. The latter primitive receives a JSON and indicates whether the remote system is genuine. The runtime abstracts these primitives using *librats* [36], a library capable of generating and verifying attestation on Intel SGX, TDX and AMD SEV-SNP. For each attester type, a matching verifier type is implemented in librats, which avoids introducing TEE architecture-specific concerns in the Wasm applications. An enclave can produce a cryptographic summary of its state through the `EREPORT` instruction. This summary, or report, includes assertions to ensure the TCB is updated and secure. In addition, it also contains code measurements of the runtime and hosted Wasm application, along with the keying materials of the current TLS session. The report is then forwarded to an Intel component known as the *quoting enclave*, which fetches a report key using the `EGETKEY` instruction to authenticate the integrity of the report. Upon successful verification, the quoting enclave generates evidence, termed a *quote* in the context of SGX, allowing the other actor (e.g. the publisher, subscriber or broker) to attest the enclave [40, 58]. As a remote attestation mechanism, we use the Intel data center attestation primitives (DCAP) [60], which avoids involving Intel during the verification of the attestation evidence, speeding up the process and allows our pub/sub system to operate without an Internet connection.

Wasm. We compiled Mosquitto using Clang with WASI-SDK [75], a toolchain that compiles C/C++ source code into Wasm for non-web environments. Mosquitto lacks support for a portable and Wasm-enabled TLS library. Therefore, we modified Mosquitto to use WolfSSL as a drop-in cryptographic library replacement. We leverage the compatibility layer of WolfSSL with OpenSSL, minimising henceforth the required code changes. Further, we disabled some of Mosquitto's system interaction layers (e.g., signals, dynamic library loading) as not supported by WASI. Turning off these features allowed a smooth embedding of Mosquitto for Wasm into the SGX enclave. The adaptation of Mosquitto required changes to 610 SLOC (in C source/header files), which accounts for 1.2% of the total codebase.

Securing the file system. Our proposal relies on the trusted file system of Twine, which is backed by the Intel protected file system (IPFS). IPFS uses AES-GCM for encryption, taking advantage of hardware acceleration from the CPU. Files are encrypted into a Merkle tree structure of 4 KiB nodes and stored on the untrusted file system, with each node securing its children through encryption keys and tags. When the enclave requests data, IPFS decrypts tree nodes within the shielded memory of the TEE, ensuring the information stored on the untrusted file system remains confidential and unaltered. In our proof of concept, IPFS automatically generates the keys based on enclave signatures and processor-specific keys. For more sophisticated use cases, such as fault tolerance, Wasm programs can provide their encryption keys, which can be obtained from an external key management system.

Supporting the cloud-edge continuum. We opted for Intel SGX for hosting our proof of concept because Twine supports all the trusted primitives outlined in Section 4.3, which are necessary for running our proposal. While alternative implementations are possible on other TEE architectures like Intel TDX and AMD SEV-SNP using the runtime WAMR [71], or Arm TrustZone using WaTZ [42], these options would require additional effort because they currently support only some of the trusted primitives we rely on. For example, WaTZ enhances Arm TrustZone with attestation capabilities but does not offer file persistence or a complete socket API. In the case of WAMR, its mechanisms for attestation have yet to be tested on AMD platforms. Nonetheless, we are confident that with some extensions to these runtimes, our prototype can be adapted to work with a variety of TEE architectures, enabling an agnostic approach for trustworthy pub/sub systems.

6 Evaluation

Our evaluations assess several aspects of our proof of concept. We intend to answer the following questions: *1)* What performance costs are associated with the use of Wasm, SGX, and the creation of attestation evidence on the broker during the connection process? *2)* How does our solution scale when increasing message throughput? *3)* How does our solution scale with a growing number of publishers? To answer these questions, we measure the connection times with different peers (§6.1), stress the broker with a high volume of messages (§6.2), and grow the number of publishers while observing the resulting latency (§6.3).

Scope. Our evaluation is deliberately centred on the broker component, which is the core and critical part of distributed pub/sub architectures. As the broker orchestrates the message flow among publishers and subscribers, it largely influences the scalability of the system. Our analysis examines the broker in different variants: native execution without SGX (the baseline), Wasm outside SGX, and Wasm-SGX, running within SGX. To ensure a consistent comparison, publishers and subscribers are executed in their native environment without SGX. While our focus remains on the broker to evaluate performance benchmarks, a fully secure system requires that all entities operate within TEEs to protect the confidentiality of data, including publishers and subscribers.

Experimental setup. The broker runs on a 16-core Intel Xeon Gold 6326 (2.9 GHz), running Ubuntu 20.04, SGX2 (EPC of 64 GiB) with SGX driver v1.41 (DCAP) and SDK v2.20. The publishers and subscribers are executed on an 8-core Intel Xeon E3-1270 v6 (3.8 GHz), running Ubuntu 20.04. These two machines are connected through a 1 Gbit/s switch. We modified Mosquitto v2.0.15 and WolfSSL v5.5.3, and compiled them using Clang 10 at

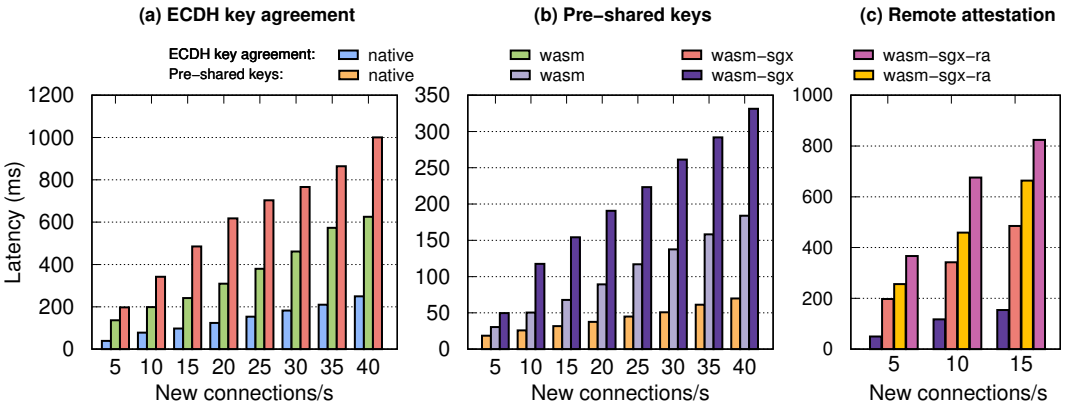


Figure 3 Latency for each new connection at varying connections per second.

maximum optimisation. For Wasm benchmarks, we initially compiled the application into Wasm bytecode with Clang, and used ahead-of-time compilation into assembly using WAMR's compiler at level size 1, as required for SGX. Time is measured using Linux's monotonic clock, including enclave exit and re-entry times (22 µs). The MQTT broker is configured to match the minimum QoS offered, i.e. fire-and-forget mode.[1] Our implementation is open-source, and instructions to reproduce our experiments are available on GitHub [45].

6.1 Establishing new connections

Our initial evaluation measures the average latency incurred when a varying number of peers attempt to establish new connections with a broker. In this setup, we instruct a certain number of peers (depicted on the x-axis) to start a new connection every second. The y-axis displays the resulting latency. Figure 3a depicts the latency for connections made using TLS handshakes with Elliptic-curve Diffie-Hellman (ECDH) as a key agreement protocol. Figure 3b shows the latency when using a pre-shared key (PSK), effectively bypassing asymmetric cryptography. The latter approach is applicable when a peer has previously established a connection with the broker and already has a mutually agreed-upon secret key.

Across both (ECDH and PSK) scenarios, latency scales linearly for every variant as the number of peers connecting per second increases from 5 to 40. We observe higher latencies with Wasm (using ECDH) over native execution with an overhead of 2.57×, due to Wasm's performance constraints relative to native execution [30]. Additionally, Wasm-SGX (using ECDH) exhibits further overhead (4.72×) compared to native because of the enclave, which is expected given the added security benefits. A contributing source of this added overhead is the need for frequent switching between the enclave's secure and standard execution modes (`OCALLs`). This is particularly evident during operations that initialise sockets and manage peer communication in the TLS handshake. For instance, using a non-blocking socket API requires constant polling when awaiting client responses. Although such overhead is also present without SGX, it is accentuated due to the enclave transitions. We point out that the data remains confidential upon leaving the enclave, as the TLS protocol operates within a secure environment. More generally, SGX also performs slower than standard execution due to security mechanisms introduced in the microcode [68].

[1] MQTT supports two other operating modes, respectively delivering the message at least once with confirmation required and delivering it exactly once. Our proposal avoids the overheads of a four-step handshake while guaranteeing both the integrity and authenticity of the involved peers.

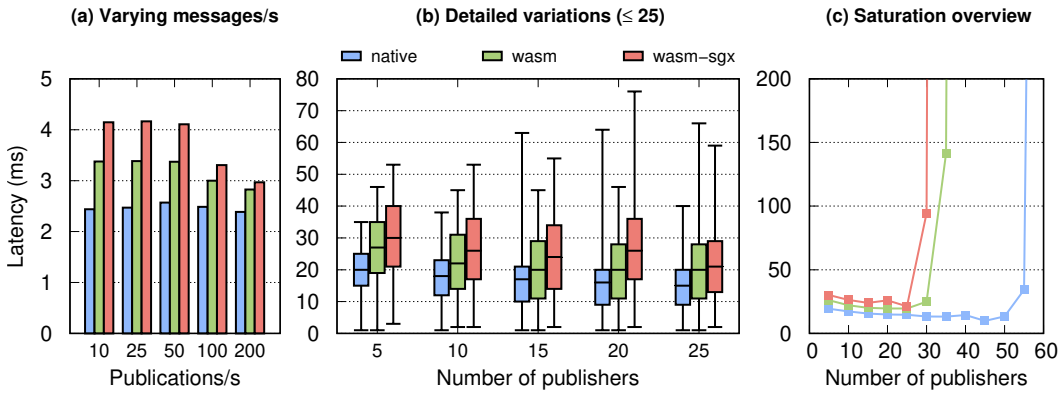


Figure 4 (a) shows latency with varying messages per second, (b) and (c) depict latency as publisher count grows.

Figure 3c explores the impact of incorporating broker attestation evidence in the TLS handshake. The overhead associated with Wasm-SGX (using ECDH) with attestation stands at 1.87×, whereas Wasm-SGX (using PSK) shows a 4.33× overhead when compared to their counterparts without attestation. Notably, the system saturates when exceeding 15 new ECDH connections per second, while PSK performance remains stable. The added latency is primarily caused by the asymmetric operations involved in evidence generation and signing. To enhance scalability and manage this overhead over a larger set of peers, Mosquitto offers a feature called *broker bridge* that can distribute the workload across multiple brokers, although an in-depth analysis of this feature is left for future work.

6.2 Messages throughput

In Figure 4a, we evaluate the system's throughput by focusing on message delivery scalability through a specific broker. More specifically, we measure the average latency of delivering a set amount of messages per second (plotted on the x-axis) with a single publisher and subscriber. The resulting latencies are reported on the y-axis. Each test case spans 60 seconds and includes a 16 kB payload of random data.

Latency for the native variant demonstrates a slight rise at a rate of 50 messages per second, followed by a minor decline at 200 messages per second. In contrast, we note a more pronounced decrease in latency for the Wasm and Wasm-SGX variants as the rate of publications increases. To understand this behaviour, we instrumented the Wasm runtime to monitor the enclave's outgoing calls (`OCALLs`) and analysed the invoked POSIX functions. Enclave transitions typically incur performance costs, such as the intrinsic SGX transition and the copy of message buffers into secure memory. We discovered that POSIX functions related to network message transactions (i.e., `recvfrom` and `send`) were called less frequently as the messages-per-second rate increased. For instance, at a rate of 10 messages per second, the frequencies of `recvfrom` and `send` are 2601 and 1411, respectively, amounting to an average of 260 and 141 per individual message. Conversely, at a higher rate of 200 messages per second, these frequencies adjust to 49 156 and 23 498, respectively, meaning a smaller average of 246 and 117 per message. As these observations appear to be caused by the behaviour of Mosquitto's broker itself, we did not conduct additional investigations.

Overall, the Wasm and Wasm-SGX variants show performance slowdown factors of 1.30× and 1.55×, respectively. Despite these factors, the system scales well and offers enhanced security and portability benefits.

6.3 Scaling the publishers

As our final experiment, we assess system scalability by increasing the number of publishers plotted on the x-axis, to observe the impact on message delivery latency, shown on the y-axis. The number of subscribers is held constant at 25, with each publisher sending 5 messages per second. Each test case lasts 60 seconds and uses a payload of 16 kB of random data.

Figure 4b reveals that all the variants follow a similar trend. Latency generally remains stable but decreases slightly as the number of publishers increases. The slowdown of Wasm compared to native execution is 1.31×, while for Wasm-SGX against native variant is 1.56×. Our analysis further investigates the point at which the system begins to saturate, thereby affecting its responsiveness. Figure 4c highlights this limit for each variant, indicating system saturation when exceeding 25 publishers for Wasm-SGX, followed by Wasm at 30 publishers. The native variant only reaches its limit beyond 55 publishers. Similarly to the first experiment, we could use Mosquitto's broker bridge feature to handle more publishers. This would increase the overall capacity and reduce the load of individual brokers.

7 Conclusion

Recent evolution in TEEs by leading CPU manufacturers highlights the growing trend in executing software within untrusted environments while processing increasingly sensitive data. The pub/sub model stands out as an effective mechanism to scale and distribute computations across varied architectures, and Wasm emerges as a suitable common environment for such tasks. However, a gap remains in establishing standardised protocols and tools that leverage the rapid research breakthroughs in trusted execution within the cloud-edge continuum.

Addressing this, we propose a secure and attested pub/sub system compatible with most of the state-of-the-art TEEs. We achieve this by using Wasm and trusted runtimes running in these TEEs. To incorporate mutual attestation in network communications, we suggest enhancing the TLS protocol. This modification embeds attestation evidence through extension points in the TLS handshake, preserving the original standard.

Our evaluation demonstrates that the security and portability improvements introduced by our approach effectively balance the additional overheads, mostly related to the TEEs. Moreover, leveraging Mosquitto's capability to distribute brokers among peers can further optimise the system, leading to a scalable and secure architecture suitable for large and real-world applications. Our implementation is freely distributed as an open-source project [45].

References

1 ISO/IEC 11889-1:2015(E). Information technology – trusted platform module library – part 1: Architecture. Standard, International Organization for Standardization, 2015.

2 A. K. M. Mubashwir Alam and Keke Chen. Making your program oblivious: a comparative study for side-channel-safe confidential computing. *CoRR*, abs/2308.06442, 2023. doi:10.48550/ARXIV.2308.06442.

3 Amazon. Pub/sub messaging, 2023. URL: https://aws.amazon.com/pub-sub-messaging.

4 AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. *White Paper*, jan 2020.

5 Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. An efficient implementation of trusted channels based on openssl. In Shouhuai Xu, Cristina Nita-Rotaru, and Jean-Pierre Seifert, editors, *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, STC 2008, Alexandria, VA, USA, October 31, 2008*, pages 41–50. ACM, 2008. doi:10.1145/1456455.1456462.

6 Sergei Arnautov, Andrey Brito, Pascal Felber, Christof Fetzer, Franz Gregor, Robert Krahn, Wojciech Ozga, André Martin, Valerio Schiavoni, Fábio Silva, Marcus Tenorio, and Nikolaus Thummel. PubSub-SGX: Exploiting trusted execution environments for privacy-preserving publish/subscribe systems. In *37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018*, pages 123–132. IEEE Computer Society, 2018. doi:10.1109/SRDS.2018.00023.

7 N. Asokan, Valtteri Niemi, and Kaisa Nyberg. Man-in-the-middle in tunnelled authentication protocols. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols, 11th International Workshop, Cambridge, UK, April 2-4, 2003, Revised Selected Papers*, volume 3364 of *Lecture Notes in Computer Science*, pages 28–41. Springer, 2003. doi:10.1007/11542322_6.

8 Pierre-Louis Aublin, Florian Kelbert, Dan O'Keffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. TaLoS: Secure and transparent TLS termination inside SGX enclaves. Technical report, Department of Computing, Imperial College London, 2017. URL: https://www.doc.ic.ac.uk/research/technicalreports/2017/DTRS17-5.pdf.

9 NorazahAbd Aziz, Nur Izura Udzir, and Ramlan Mahmod. Extending TLS with mutual attestation for platform integrity assurance. *J. Commun.*, 9(1):63–72, 2014. doi:10.12720/JCM.9.1.63-72.

10 Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica, and Etienne Rivière. Efficient and confidentiality-preserving content-based publish/subscribe with prefiltering. *IEEE Trans. Dependable Secur. Comput.*, 14(3):308–325, 2017. doi:10.1109/TDSC.2015.2449831.

11 Stefano Berlato, Umberto Morelli, Roberto Carbone, and Silvio Ranise. End-to-end protection of IoT communications through cryptographic enforcement of access control policies. In Shamik Sural and Haibing Lu, editors, *Data and Applications Security and Privacy XXXVI – 36th Annual IFIP WG 11.3 Conference, DBSec 2022, Newark, NJ, USA, July 18-20, 2022, Proceedings*, volume 13383 of *Lecture Notes in Computer Science*, pages 236–255. Springer, 2022. doi:10.1007/978-3-031-10684-2_14.

12 Cristian Borcea, Arnab Deb Gupta, Yuriy Polyakov, Kurt Rohloff, and Gerard W. Ryan. PICADOR: end-to-end encrypted publish-subscribe information distribution with proxy re-encryption. *Future Gener. Comput. Syst.*, 71:177–191, 2017. doi:10.1016/J.FUTURE.2016.10.013.

13 Sébanjila Kevin Bukasa, Ronan Lashermes, Hélène Le Bouder, Jean-Louis Lanet, and Axel Legay. How TrustZone could be bypassed: Side-channel attacks on a modern system-on-chip. In Gerhard P. Hancke and Ernesto Damiani, editors, *Information Security Theory and Practice – 11th IFIP WG 11.2 International Conference, WISTP 2017, Heraklion, Crete, Greece, September 28-29, 2017, Proceedings*, volume 10741 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2017. doi:10.1007/978-3-319-93524-9_6.

14 Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *CoRR*, abs/2303.15540, 2023. doi:10.48550/ARXIV.2303.15540.

15 Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016. URL: http://eprint.iacr.org/2016/086.

16 Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 857–874. USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

17 Enarx. Confidential computing with WebAssembly, 2019. URL: https://enarx.dev.

18 Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003. doi:10.1145/857076.857078.

19 Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Comput. Surv.*, 54(6):126:1–126:36, 2022. doi:10.1145/3456631.

20 Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305. ACM, 2017. doi:10.1145/3132747.3132782.

21 Sarah Abdelwahab Gaballah, Christoph Coijanovic, Thorsten Strufe, and Max Mühlhäuser. 2PPS – publish/subscribe with provable privacy. In *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*, pages 198–209. IEEE, 2021. doi:10.1109/SRDS53918.2021.00028.

22 Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, and N. Asokan. Beyond secure channels. In Peng Ning, Vijay Atluri, Shouhuai Xu, and Moti Yung, editors, *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing, STC 2007, Alexandria, VA, USA, November 2, 2007*, pages 30–40. ACM, 2007. doi:10.1145/1314354.1314363.

23 Kenneth A. Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In Ari Juels, Gene Tsudik, Shouhuai Xu, and Moti Yung, editors, *Proceedings of the 1st ACM Workshop on Scalable Trusted Computing, STC 2006, Alexandria, VA, USA, November 3, 2006*, pages 21–24. ACM, 2006. doi:10.1145/1179474.1179481.

24 Google. Pub/sub, 2023. URL: https://cloud.google.com/pubsub.

25 Christian Göttel, Pascal Felber, and Valerio Schiavoni. Developing secure services for IoT with OP-TEE: A first look at performance and usability. In José Pereira and Laura Ricci, editors, *Distributed Applications and Interoperable Systems – 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11534 of *Lecture Notes in Computer Science*, pages 170–178. Springer, 2019. doi:10.1007/978-3-030-22496-7_11.

26 Franz Gregor, Wojciech Ozga, Sébastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, André Martin, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. Trust management as a service: Enabling trusted execution in the face of byzantine stakeholders. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 – July 2, 2020*, pages 502–514. IEEE, 2020. doi:10.1109/DSN48063.2020.00063.

27 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi:10.1145/3062341.3062363.

28 Intel. Intel software guard extensions remote attestation end-to-end example, jul 2018. URL: https://software.intel.com/content/www/us/en/develop/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.html.

29 Mihaela Ion, Giovanni Russello, and Bruno Crispo. Design and implementation of a confidentiality and access control solution for publish/subscribe systems. *Comput. Networks*, 56(7):2014–2037, 2012. doi:10.1016/J.COMNET.2012.02.013.

30 Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 107–120. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/jangda.

31 André Joaquim, Miguel L. Pardal, and Miguel Correia. Vulnerability-tolerant transport layer security. 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 28:1–28:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.OPODIS.2017.28.

32 Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: EPID provisioning and attestation services. *White Paper*, 1(1-10):119, 2016. URL: https://cdrdv2-public.intel.com/671370/ww10-2016-sgx-provisioning-and-attestation-final.pdf.

33 Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *CoRR*, abs/1801.05863, 2018. arXiv:1801.05863.

34 Hugo Krawczyk. SIGMA: The "SIGn-and-MAc" approach to authenticated Diffie-Hellman and its use in the IKE-protocols. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 400–425. Springer, 2003. doi:10.1007/978-3-540-45146-4_24.

35 Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 38:1–38:16. ACM, 2020. doi:10.1145/3342195.3387532.

36 Librats. Low level attester and verifier drivers for multiple TEEs, 2022. URL: https://github.com/inclavare-containers/librats.

37 Roger A. Light. Mosquitto: server and client implementation of the MQTT protocol. *J. Open Source Softw.*, 2(13):265, 2017. doi:10.21105/JOSS.00265.

38 Lukas Malina, Gautam Srivastava, Petr Dzurenda, Jan Hajny, and Radek Fujdiak. A secure publish/subscribe protocol for internet of things. In *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*, pages 75:1–75:10. ACM, 2019. doi:10.1145/3339252.3340503.

39 Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 143–158. IEEE Computer Society, 2010. doi:10.1109/SP.2010.17.

40 Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. Attestation mechanisms for trusted execution environments demystified. In David M. Eyers and Spyros Voulgaris, editors, *Distributed Applications and Interoperable Systems: 22nd IFIP WG 6.1 International Conference, DAIS 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, volume 13272 of *Lecture Notes in Computer Science*, pages 95–113. Springer, 2022. doi:10.1007/978-3-031-16092-9_7.

41 Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. TWINE: An embedded trusted runtime for WebAssembly. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 205–216. IEEE, 2021. doi:10.1109/ICDE51399.2021.00025.

42 Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. WATZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*, pages 1177–1189. IEEE, 2022. doi:10.1109/ICDCS54860.2022.00116.

43 Microsoft. Publisher-subscriber pattern, 2023. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber.

44 Mozilla. Standardizing WASI: A system interface to run WebAssembly outside the web, mar 2019. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

45 Jämes Ménétrey. A holistic approach for trustworthy distributed systems with WebAssembly and TEEs: code and benchmarks, 2023. URL: https://github.com/JamesMenetrey/unine-opodis2023.

46 Mohamed Nabeel, Stefan Appel, Elisa Bertino, and Alejandro P. Buchmann. Privacy preserving context aware publish subscribe systems. In Javier López, Xinyi Huang, and Ravi S. Sandhu, editors, *Network and System Security – 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, volume 7873 of *Lecture Notes in Computer Science*, pages 465–478. Springer, 2013. doi:10.1007/978-3-642-38631-2_34.

47 Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. Everything you should know about intel SGX performance on virtualized systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(1):5:1–5:21, 2019. doi:10.1145/3322205.3311076.

48 Arto Niemi, Vasile Adrian Bogdan Pop, and Jan-Erik Ekberg. Trusted sockets layer: A TLS 1.3 based trusted channel protocol. In Nicola Tuveri, Antonis Michalas, and Billy Bob Brumley, editors, *Secure IT Systems – 26th Nordic Conference, NordSec 2021, Virtual Event, November 29-30, 2021, Proceedings*, volume 13115 of *Lecture Notes in Computer Science*, pages 175–191. Springer, 2021. doi:10.1007/978-3-030-91625-1_10.

49 Arto Niemi, Sampo Sovio, and Jan-Erik Ekberg. Towards interoperable enclave attestation: Learnings from decades of academic work. In *31st Conference of Open Innovations Association, FRUCT 2022, Helsinki, Finland, April 27-29, 2022*, pages 189–200. IEEE, 2022. doi:10.23919/FRUCT54823.2022.9770907.

50 Emanuel Onica, Pascal Felber, Hugues Mercier, and Etienne Rivière. Confidentiality-preserving publish/subscribe: A survey. *ACM Comput. Surv.*, 49(2):27:1–27:43, 2016. doi:10.1145/2940296.

51 A Paverd. *Enhancing communication privacy using trustworthy remote entities*. PhD thesis, University of Oxford, 2015.

52 Jinglei Pei, Yuyang Shi, Qingling Feng, Ruisheng Shi, Lina Lan, Shui Yu, Jinqiao Shi, and Zhaofeng Ma. An efficient confidentiality protection solution for pub/sub system. *Cybersecur.*, 6(1):34, 2023. doi:10.1186/S42400-023-00165-W.

53 Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. Secure content-based routing using Intel software guard extensions. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 – 16, 2016*, page 10. ACM, 2016. doi:10.1145/2988336.2988346.

54 Ivan Puddu, Moritz Schneider, Daniele Lain, Stefano Boschetto, and Srdjan Capkun. On (the lack of) code confidentiality in trusted execution environments. *CoRR*, abs/2212.07899, 2022. arXiv:2212.07899, doi:10.48550/ARXIV.2212.07899.

55 Eric Rescorla. Keying material exporters for transport layer security (TLS). RFC 5705, mar 2010. doi:10.17487/RFC5705.

56 Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. TLS encrypted client hello. Internet-Draft draft-ietf-tls-esni-16, Internet Engineering Task Force, apr 2023. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-ietf-tls-esni/16/.

57 Peter Saint-Andre and Jeff Hodges. Representation and verification of domain-based application service identity within internet public key infrastructure using X.509 (PKIX) certificates in the context of transport layer security (TLS). RFC 6125, mar 2011. doi:10.17487/RFC6125.

58 Muhammad Usama Sardar, Rasha Faqeh, and Christof Fetzer. Formal foundations for Intel SGX data center attestation primitives. In Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony, editors, *Formal Methods and Software Engineering – 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, volume 12531 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2020. doi:10.1007/978-3-030-63406-3_16.

59 Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. Demystifying attestation in Intel trust domain extensions via formal verification. *IEEE Access*, 9:83067–83079, 2021. doi:10.1109/ACCESS.2021.3087421.

60 Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for Intel SGX with Intel data center attestation primitives. *White paper*, page 12, 2018. URL: https://cdrdv2-public.intel.com/671314/intel-sgx-support-for-third-party-attestation.pdf.

61 Carlos Segarra, Ricard Delgado-Gonzalo, and Valerio Schiavoni. MQT-TZ: Hardening IoT brokers using ARM TrustZone : (practical experience report). In *International Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*, pages 256–265. IEEE, 2020. doi:10.1109/SRDS51746.2020.00033.

62 Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. Establishing mutually trusted channels for remote sensing devices with trusted execution environments. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 – September 01, 2017*, pages 7:1–7:10. ACM, 2017. doi:10.1145/3098954.3098971.

63 Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the scalability of platform attestation. In Shouhuai Xu, Cristina Nita-Rotaru, and Jean-Pierre Seifert, editors, *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, STC 2008, Alexandria, VA, USA, October 31, 2008*, pages 1–10. ACM, 2008. doi:10.1145/1456455.1456457.

64 Frederic Stumpf, Omid Tafreschi, Patrick Röder, Claudia Eckert, et al. A robust integrity reporting protocol for remote attestation. In *Proceedings of the Workshop on Advances in Trusted Computing (WATC)*, page 65, 2006.

65 Antero Taivalsaari, Tommi Mikkonen, and Cesare Pautasso. Towards seamless IoT device-edge-cloud continuum: Software architecture options of IoT devices revisited. In Maxim Bakaev, In-Young Ko, Michael Mrissa, Cesare Pautasso, and Abhishek Srivastava, editors, *ICWE 2021 Workshops – ICWE 2021 International Workshops, BECS and Invited Papers, Biarritz, France, May 18-21, 2021*, volume 1508 of *Communications in Computer and Information Science*, pages 82–98. Springer, 2021. doi:10.1007/978-3-030-92231-3_8.

66 Trusted Computing Group. *DICE Attestation Architecture*, 2021. URL: https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-r23-final.pdf.

67 Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. COBRA: dynamic proactive secret sharing for confidential BFT services. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 1335–1353. IEEE, 2022. doi:10.1109/SP46214.2022.9833658.

68 Sébastien Vaucher, Valerio Schiavoni, and Pascal Felber. Short paper: Stress-SGX: Load and stress your enclaves for fun and profit. In Andreas Podelski and François Taïani, editors, *Networked Systems – 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers*, volume 11028 of *Lecture Notes in Computer Science*, pages 358–363. Springer, 2018. doi:10.1007/978-3-030-05529-5_24.

69 Paul Georg Wagner, Pascal Birnstill, and Jürgen Beyerer. Establishing secure communication channels using remote attestation with TPM 2.0. In Konstantinos Markantonakis and Marinella Petrocchi, editors, *Security and Trust Management – 16th International Workshop, STM 2020, Guildford, UK, September 17-18, 2020, Proceedings*, volume 12386 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2020. doi:10.1007/978-3-030-59817-4_5.

70 Kevin Walsh and John Manferdelli. Mechanisms for mutual attested microservice communication. In Ashiq Anjum, Alan Sill, Geoffrey C. Fox, and Yong Chen, editors, *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, pages 59–64. ACM, 2017. doi:10.1145/3147234.3148102.

71 WAMR. WebAssembly micro runtime, 2019. URL: https://github.com/bytecodealliance/wasm-micro-runtime.

72 Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pages 3940–3947. IEEE, 2002. doi:10.1109/HICSS.2002.994531.

73 Shuran Wang, Dahan Pan, Runhan Feng, and Yuanyuan Zhang. Magikcube: Securing cross-domain publish/subscribe systems with enclave. In *20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2021, Shenyang, China, October 20-22, 2021*, pages 147–154. IEEE, 2021. doi:10.1109/TRUSTCOM53373.2021.00037.

74 Wenwen Wang. How far we've come – A characterization study of standalone WebAssembly runtimes. In *IEEE International Symposium on Workload Characterization, IISWC 2022, Austin, TX, USA, November 6-8, 2022*, pages 228–241. IEEE, 2022. doi:10.1109/IISWC55918.2022.00028.

75 WASI-SDK. WASI-enabled WebAssembly C/C++ toolchain, 2019. URL: https://github.com/WebAssembly/wasi-sdk.

76 Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. doi:10.14722/ndss.2019.23068.

77 Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In Steven D. Galbraith, Giovanni Russello, Willy Susilo, Dieter Gollmann, Engin Kirda, and Zhenkai Liang, editors, *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*, pages 73–85. ACM, 2019. doi:10.1145/3321705.3329820.

78 Yue Yu, Huaimin Wang, Bo Liu, and Gang Yin. A trusted remote attestation model based on trusted computing. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013*, pages 1504–1509. IEEE Computer Society, 2013. doi:10.1109/TRUSTCOM.2013.183.

Recoverable and Detectable Self-Implementations of Swap

Tomer Lev Lehman
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Danny Hendler
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Abstract

Recoverable algorithms tolerate failures and recoveries of processes by using non-volatile memory. Of particular interest are *self-implementations* of key operations, in which a recoverable operation is implemented from its non-recoverable counterpart (in addition to reads and writes).

This paper presents two self-implementations of the *swap* operation. One works in the *system-wide failures* model, where all processes fail and recover together, and the other in the *independent failures* model, where each process crashes and recovers independently of the other processes.

Both algorithms are wait-free in crash-free executions, but their recovery code is blocking. We prove that this is inherent for the independent failures model. The impossibility result is proved for implementations of *distinguishable* operations using *interfering* functions, and in particular, it applies to a recoverable self-implementation of swap.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Multi-core algorithms, persistent memory, non-volatile memory, recoverable objects, detectablitly

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.24

Related Version *Full Version*: `https://arxiv.org/abs/2308.03485`

Funding Partially supported by the Israel Science Foundation (grant 22/1425).

1 Introduction

Recent years have seen a rising interest in the failure-recovery model for concurrent computing. This model captures an unstable system, where processes may crash and recover. Two variants of the model have been considered. In the *system-wide failures* model (also called the *global-crash* model), all processes fail simultaneously and a single process is responsible for the recovery of the whole system. In the *independent failures* model (also called the *individual-crash* model), each process can incur a crash independently of other processes and recovers independently. *Recoverable* algorithms, tolerating failures and recoveries, have been presented for various concurrent data structures, for both the system-wide failures model [9, 11, 14, 17, 24, 34, 37, 40] and the independent failures model [2, 4, 9, 34, 37].

The correctness of a recoverable algorithm can be specified in several ways. *Durable Linearizability* [26] intuitively requires linearizability [23] of all operations that survive the crashes. *Detectability* [17] ensures that upon recovery, it is possible to infer whether the failed operation took effect or not and, in the former case, obtain its response. *Nesting-safe Recoverable Linearizability* (NRL) [2], originally defined for the independent failures model, ensures detectability and linearizability.

NRL is defined in a way that facilitates the nesting of recoverable objects. By providing implementations of NRL primitive operations on base objects, a programmer can combine several of these primitives to create recoverable implementations of complex higher-level objects and algorithms. This level of abstraction can be helpful in the adoption of recoverable algorithms.

To facilitate high-level implementations of complex NRL objects, it is helpful to introduce implementations of low-level NRL objects. An attractive approach to designing low-level NRL objects is through *self-implementations* [37], in which a recoverable operation is implemented with instances of the *same* primitive operation, possibly with additional reads and writes on shared variables. This approach ensures that when using the recoverable version of an operation, the system must only support its hardware-implemented primitive counterpart.

NRL self-implementations already exist for various primitives, including *read*, *write*, *test&set*, and *compare&swap* [2, 4, 27], as well as *fetch&add* [37]. A universal construction using NRL read, write and *compare&swap* objects [4] builds upon previously-introduced self-implementations of NRL objects to take any concurrent program with read write and CAS, and make it recoverable while adding only constant computational overhead.

This paper presents the first NRL self-implementations of *swap*, for both the system-wide and the independent failures models. Swap is a widely-used primitive that is employed by concurrent algorithms. Our implementations borrow ideas from the *recoverable mutual exclusion (RME)* [20] algorithms of [19, 29], which use a similar approach to overcome swap failures. Unlike these algorithms, however, our implementations are also challenged with the task of satisfying wait-freedom and linearizability. Both our algorithms are wait-free in crash-free executions, while the recovery code in both is blocking.

We also prove the impossibility of implementing a class of *distinguishable operations* using a set of *interfering functions* [22] in a recoverable *lock-free* fashion in the independent failures model. In particular, this result applies to self-implementations of swap, but it also holds for, e.g., implementing swap using a combination of fetch-and-add and swap. Other distinguishable operations to which this proof applies are the *deque* of a queue object and the *pop* of a stack object. Our impossibility result unifies and extends specialized results for self-implementations of *test&set* [2] and *fetch&add* [37]. Another related impossibility result addresses recoverable consensus in the independent failures model [18].

To summarize, our contributions are the following:

- A recoverable detectable self-implementation of swap in the system-wide failures model.
- A recoverable detectable self-implementation of swap in the independent failures model.
- An impossibility proof for implementations of distinguishable operations using interfering functions in the independent failures model.

Related Work

Several correctness conditions and definitions, in addition to those previously mentioned, exist for recoverable algorithms. *Strict linearizability* [1] requires operations interrupted by a failure to take effect either before the failure or not at all. A relaxed condition called *persistent atomicity* [21], defined for message-passing systems, allows an operation to take effect even after a failure, before the next operation invocation of the same process. *Recoverable linearizability* [6] builds on the definition of persistent atomicity, but it also ensures *locality* – the desirable property that an execution involving multiple objects is correct if and only if its projection onto each individual object is correct [23].

Ben-David, Friedman and Wei [5] define a hierarchy based on the sets of execution histories that are allowed by each of the above definitions under the individual-crash model, in which the same processes are allowed to be invoked following a crash. In this hierarchy,

NRL is not implied by the above definitions, but it does overlap with some of them. In particular, any strictly linearizable or persistent atomic history completion in which no operation was removed is also consistent with NRL. Since NRL only requires the completion of an operation before the next one by the same process, a history may be NRL consistent even if some process executes an entire operation after the crash before another process completes its operation which was started before the crash. Such a history is not strictly linearizable. However, it is still persistently atomic. In fact, persistent atomicity is strictly stronger than NRL under this history completions interpretation.

Detectable sequence specifications (DSS) [34] also formalizes the notion of detectability. DSS is more portable and less reliant on system assumptions in comparison to NRL, but it delegates the responsibility for nesting to the application code.

Several papers introduce general mechanisms to port existing algorithms and make them persistent, e.g., by using transactional memory [7,11,25,39], universal constructions [6,10,12], or for specific families of algorithms [4,13,16]. Most of these transformations use strong primitives such as *CAS* while their non-recoverable counterparts may use weaker primitives, in terms of their consensus number [22]. We believe future research may use our self-implementation of swap to extend general constructions such as [4], mentioned above, to programs that also use swap as a primitive. In addition, using NRL self-implementations can ensure the ability to implement higher-level objects using the same primitives as would be used in a non-recoverable setting. In contrast, other works [9,14,17,34,35,40] mostly rely on strong primitives such as *CAS*.

Other papers present hand-crafted persistent implementations of specific data structures, e.g., [17,38,41,42]. In contrast to these implementations, our algorithms provide a recoverable counterpart to an atomic primitive operation, which we believe can later be used in various other implementations of recoverable algorithms.

Our algorithm for the independent failures model uses an RME lock such as the one presented by Golab and Ramaraju [20], which uses only reads and writes. Additionally, there is a long line of research on RME, solving several other aspects such as abortability, FCFS, and more (see, e.g., [8,28,30–33]).

2 Model and Definitions

Since this work deals with a recoverable version of a primitive object, we assume a simplified version of the NRL system model [2], which does not capture nesting. There are n asynchronous *processes* $p_1, \ldots, p_n$, which communicate by applying atomic *primitive* read, write and read-modify-write operations to *base objects*. The state of each process consists of non-volatile *shared variables*, which serve as base objects, as well as volatile *local variables*.

We first describe the *independent failures* model, in which each process can incur a *crash-failure* (or simply a *crash*) at any point during the execution independently of other processes. A crash resets all of its local variables to arbitrary values but preserves the values of all non-volatile variables.

A process p *invokes an operation Op* on an object by performing an *invocation step*. *Op completes* by executing a *response step*, in which *the response of OP is stored to a local volatile variable of p*. It follows that the response value is lost if p incurs a crash before *persisting* it, that is, before writing it to a non-volatile variable. Operation *Op* is *pending* if it was invoked but was not yet completed; a process has at most one pending operation.

Each *recoverable operation Op* has an associated *recovery procedure Op*.RECOVER, which is responsible for completing *Op* upon recovery from a crash. If the object supports a *single* recoverable operation, then its recovery procedure is simply named RECOVER. Following

a crash of process p that occurs when p has a pending recoverable operation instance, the system eventually resurrects process p by invoking the recovery procedure of the recoverable operation that was pending when p failed. This is represented by a *recovery step for p*.

Formally, a *history* H is a sequence of *steps*. There are four types of steps:

1. An *invocation step*, denoted (INV, p, O, Op), represents the invocation by process p of operation Op on object O.
2. A *response step* s, denoted (RES, p, O, Op, ret), represents the completion by process p of operation Op invoked on object O by some (invocation) step s' of p, with response ret being written to a local variable of p; *s is the response step that matches s'*. An operation Op can be completed either normally or when, following one or more crashes, the execution of Op.RECOVER is completed.
3. A *crash step* s, denoted $(CRASH, p)$, represents the crash of process p. We call the recoverable operation Op of p that was pending when the crash occurred the *crashed operation of s*. $(CRASH, p)$ may also occur when the recovery procedure Op.RECOVER is executed for recovering an operation of p and we say that Op is the crashed operation of s also in this case.
4. A *recovery step s for process p*, denoted (REC, p), is the only step by p that is allowed to follow a $(CRASH, p)$ step s'. It represents the resurrection of p by the system, in which it invokes Op.RECOVER, where Op is the crashed operation of s'. We say that *s is the recovery step that matches s'*.

An object is *recoverable* if all its operations are recoverable. Below, we consider only histories that arise from operations on recoverable objects or atomic primitive operations.

When a recovery procedure Op.RECOVER is invoked to recover from a crash represented by step s, we assume it receives the same arguments as those with which Op was invoked when that crash occurred.

It was shown [3] that detectable algorithms for the NRL model must keep an auxiliary state that is provided from outside the operation, either via operation arguments or via a non-volatile variable accessible by it. We assume that Op.RECOVER has access to a designated per-process non-volatile variable SEQ_p, storing *the sequence number of Op*. Before p invokes an operation on the object, it increments SEQ_p.

Consider a scenario in which p incurs a crash (represented by a crash step s) immediately after a recoverable operation Op completes (either directly or through the completion of Op.RECOVER); this event is represented by a response step r. In this case, the response value is lost and, moreover, Op.RECOVER will not be invoked by the system, since the crashed operation of s is no longer Op. In general, p may therefore not be able to proceed correctly. Hence, it is sometimes required to guarantee that a recoverable operation returns only once its response value gets persisted. This is defined formally as follows.

▶ **Definition 1.** *A recoverable operation Op is* strictly recoverable *[2] if whenever it is completed, either directly or by the completion of Op.Recover, with a (RES, p, O, Op, ret) step event, ret is stored in a designated persistent variable accessed only by process p.*

A history H is *crash-free* if it contains no crash steps (hence also no recovery steps). $H|p$ denotes the sub-history of H consisting of all the steps by process p in H. $H|O$ denotes the sub-history of H consisting of all the invocation and response steps on object O in H, as well as any crash step in H, by any process p, whose crashed operation is an operation on O and the corresponding recovery step by p (if it appears in H). $H|<p, O>$ denotes the sub-history consisting of all the steps on O by p.

A crash-free sub-history $H|O$ is *well-formed*, if for all processes p, $H|<p, O>$ is a sequence of alternating, matching invocation and response steps, starting with an invocation step. A crash-free history H is *well-formed* if:

1. $H|O$ is well-formed for all objects O, and
2. each invocation event in $H|p$, except possibly the last one, is immediately followed by its matching response step.

$H|O$ is a *sequential object history* if it is an alternating series of invocations and the matching responses starting with an invocation; it may end with a pending invocation. The *sequential specification* of an object O is the set of all *legal* sequential histories over O. H is a *sequential history* if $H|O$ is a sequential object history for all objects O.

Two histories H and H' are *equivalent*, if $H|<p, O> = H'|<p, O>$ for all processes p and objects O. Given a history H, a *completion* of H is a history H' constructed from H by selecting separately, for each object O that appears in H, a subset of the operations pending on O in H and appending matching responses to all these operations, and then removing all remaining pending operations on O (if any).

An operation op_1 precedes an operation op_2 in H, denoted $op_1 <_H op_2$, if op_1's response step is before the invocation step of op_2 in H.

▶ **Definition 2** (Linearizability [23], rephrased)**.** *A finite crash-free history H is* linearizable *if it has a completion H' and a legal sequential history S such that H' is equivalent to S and $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both op_1 and op_2 appear in S, then $op_1 <_S op_2$).*

To define nesting-safe recoverable linearizability, we introduce a more general notion of well-formedness that also applies to histories that contain crash/recovery steps. For a history H, we let $N(H)$ denote the history obtained from H by removing all crash and recovery steps. A history H is *recoverable well-formed* if every crash step in $H|p$ is either p's last step in H or is followed in $H|p$ by a matching recovery step of p, and $N(H)$ is well-formed.

▶ **Definition 3** (Nesting-safe Recoverable Linearizability (NRL))**.** *A finite history H satisfies* nesting-safe recoverable linearizability *(NRL) if it is recoverable well-formed and $N(H)$ is a linearizable history. An object implementation satisfies NRL if all of its finite histories satisfy NRL.*

We build upon the above definitions also for the *system-wide failures* model, but we require that if a crash step occurs, then it occurs simultaneously for all processes whose operations crash. Formally, let $p_{i_1}, \ldots, p_{i_k}$, for $k \leq n$, be the set of processes that have a pending invocation of operation Op when a system-wide crash occurs. The crash is represented by appending the sequence $(CRASH, p_{i_1}), \ldots, (CRASH, p_{i_k})$ to the execution. During recovery, the system executes a parameterless *global recovery procedure for Op* called Op.GRECOVER, which is represented by appending the sequence $(REC, p_{i_1}), \ldots, (REC, p_{i_k})$ to the execution. Once Op.GRECOVER completes, the system resurrects each of the processes $p_{i_1}, \ldots, p_{i_k}$ to perform an *individual recovery procedure for Op*, called Op.RECOVER. New operations on the object can be invoked only after recovery ends. If Op is a single object operation, we write GRECOVER and RECOVER instead of Op.GRECOVER and Op.RECOVER, respectively.

An algorithm is *lock-free* if, whenever a set of processes take a sufficient number of steps and none of them crashes, then it is guaranteed that one of them will complete its operation. An algorithm is *wait-free*, if any process that does not incur a crash and takes a sufficient number of steps completes its operation in a finite number of its steps. A swap object supports the *swap(val)* operation, which atomically swaps the object's current value *cur* to *val* and returns *cur*. For presentation clarity, we denote from now on the primitive swap operation by (lowercase) swap and its counterpart implemented recoverable operation by (uppercase) SWAP.

3 Detectable Swap Algorithm for the System-Wide Failures Model

A key challenge to overcome when implementing a detectable swap object from read, write, and primitive swap operations is that the return values of one or more primitive swap operations may be lost upon a system-wide failure that occurs before they are persisted. These operations may have already affected the state of the swap object and, moreover, operations by other processes may have already returned the values written by these primitive operations. To ensure linearizability, the implementation must identify such operations and handle them correctly.

The return value of each SWAP operation must meet a few requirements. First, it should be the input of another SWAP operation (or the initial value of the swap object). Second, the operand swapped in by one SWAP operation can be returned by at most a single other SWAP operation. Finally, in order to maintain linearizability, if $op_1 <_H op_2$ holds, then op_1 cannot return the value that was swapped in by op_2.

Figure 1 illustrates a scenario involving six processes, $p_1, \ldots p_6$, performing eight SWAP operations, $op_0, \ldots op_7$. A system-wide crash occurs when operations op_0, op_2, op_4, op_6 have already been completed (their return values are specified in Figure 1) while operations op_1, op_4, op_5, op_7 are pending. Note that operations op_1, op_4, op_5, although not completed, have affected the global state of the swap object as their inputs are the return values of other operations, while op_7 (pending as well) might or might not have affected the object's state.

There are several ways the system may recover in order to produce a correct linearizable result. In all of them, op_1 must return 0. The remaining uncompleted operations might return different values in the following ways:

1. op_4 returns 2, op_5 returns 3, and op_7 returns 6.
2. op_7 returns 2, op_4 returns 7, and op_5 returns 3.
3. op_4 returns 2, op_7 returns 3, and op_5 returns 7.

In this example there are several possible linearizations because op_7 may be linearized in several ways since its effect on the global state is unknown. Note that for correctly recovering op_1, the operand of the very first operation, op_0, must be recorded. This is why our algorithms retain a record of all invoked operations.

We represent the order of SWAP operations as a linked list of Node structures, the end of which is pointed by a *tail* variable manipulated using primitive swaps. The list starts with a sentinel node called *headNode*, which holds the object's initial value (denoted $\perp$).

Each Node structure represents a single SWAP operation and stores a pointer *prev* to the node of its predecessor operation and the SWAP's operand *val*. The order of SWAP operations is reflected by the order of the Node structures in the list. By doing so, each Node points to the previous Node structure that represents the previous SWAP operation, hence the operation's return value will be *Node.prev.val*.

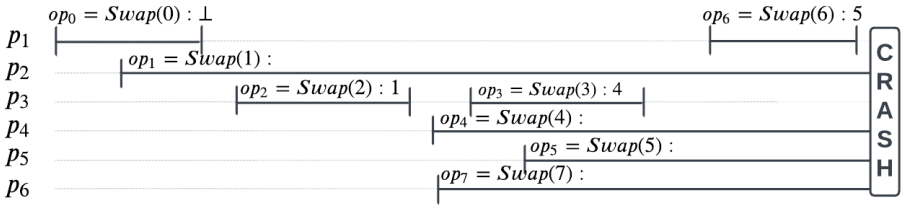


Figure 1 An example of the effect of a system-wide failure.

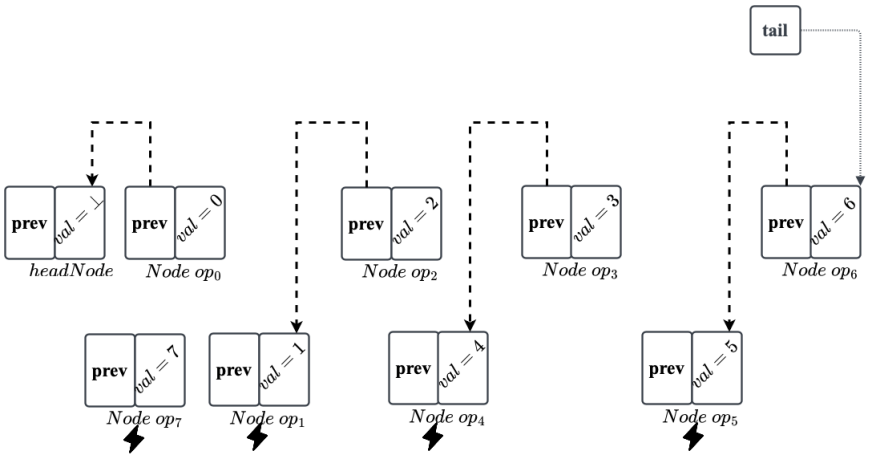


Figure 2 The fragments corresponding to Figure 1. Operations 1,4,5 crashed after swapping their Nodes to *tail* but before persisting their pointer to their predecessor Node.

A problem can occur if a process successfully swaps its Node into the list but crashes before pointing from its structure to the previous Node. This type of failure may create what is referred to as *fragments* in the list representing the SWAP operations. Thus, instead of a single complete list, crashes may result in several incomplete disconnected lists. In order to reconnect these fragments back to a complete list, our algorithms go over all previously-announced operations upon recovery and recreate a correctly ordered complete list of operations.

A similar idea was used by the recoverable mutual exclusion (RME) algorithms of Golab and Hendler [19] and Jayanti, Jayanti and Joshi [29]. These algorithms also have to reconnect the fragments of an MCS lock [36] linked-list based queue, caused by failures that occur just before or after primitive swap operations. However, our algorithms need to address two challenges that do not exist in RME algorithms. First, the SWAP operations of our algorithms are required to be wait-free whereas RME implementations are allowed to block. Second, unlike RME implementations, our algorithms are required to maintain linearizability. Specifically, the new order of list fragments, constructed during recovery, must respect the real-time order between SWAP operations.

We address these challenges by employing a *fragment ordering* scheme, which we view as the key novelty of our algorithms. The scheme encapsulates the critical steps of each SWAP operation by two vector timestamp computations. Based on the resulting timestamps, the recovery code ensures the following invariant: if a fragment A contains a Node n_A that was created after an operation associated with a Node n_B on fragment B was completed, then fragment B will be ordered after fragment A in the linked list. We formally define this fragment ordering scheme in Definition 4.

Figure 2 presents the fragments corresponding to Figure 1. We describe it next and introduce a few terms that are used later. The fragment of op_7 contains a single Node; we name such 1-size fragments *singleNodes*. The node of op_0 belongs to the single fragment that contains *headNode*; we name this fragment the *head fragment*. The nodes of op_5 and op_6 belong to the single fragment that contains the node pointed to by *tail*; we name this fragment the *tail fragment*. Fragments such as (op_2, op_1) and (op_3, op_4) that are neither singleNodes, nor head nor tail fragments are called *middle fragments*.

Algorithm 1 Recoverable detectable SWAP for system-wide failures model, code for process i; text in blue is for the independent failures algorithm only.

Define Node: struct {*val* : **Value**, *prev* : **ref** to Node, *prevExecution* : **ref** to Node, *startVts* : **vector**, *endVts* : **vector**, *seq* : **int**, *inWork* : **int**}

Initial State:

$Nodes[i] \leftarrow null$ for $i \in \{0, \ldots, n\}$
$VTS[i] \leftarrow 0$ for $i \in \{1, \ldots, n\}$
$headNode \leftarrow new(Node)$
$headNode.val \leftarrow \perp$, $headNode.seq \leftarrow 0$, $headNode.prev \leftarrow null$, $headNode.prevExecution \leftarrow null$, $headNode.inWork \leftarrow 0$.
$headNode.startVts \leftarrow collect(VTS)$, $headNode.endVts \leftarrow collect(VTS)$
$tail \leftarrow headNode$
$Nodes[0] \leftarrow headNode$

1: **procedure** SWAP(*val*) ▷ executed by process i
2: $myNode \leftarrow new(Node)$
3: $myNode.prev \leftarrow$null, $myNode.seq \leftarrow SEQ_i$, $myNode.prevExecution \leftarrow$ null, $myNode.val \leftarrow$val
4: $VTS[i] \leftarrow VTS[i] + 1$
5: $myNode.startVts \leftarrow collect(VTS)$
6: $prevExecution \leftarrow Nodes[i]$
7: $myNode.prevExecution \leftarrow prevExecution$
8: $myNode.inWork \leftarrow 1$ ▷ begin swap
9: $Nodes[i] \leftarrow myNode$ ▷ announce the operation
10: $prev \leftarrow swap(\&tail, myNode)$
11: $myNode.prev \leftarrow prev$ ▷ persisting operation
12: $myNode.endVts \leftarrow collect(VTS)$
13: $myNode.inWork \leftarrow 0$ ▷ finished operation
14: $Res_i \leftarrow myNode.prev.val$
15: **return** Res_i

When ordering middle fragments and singleNodes, the algorithm uses vector timestamps for maintaining linearizability. As an example, consider a linked list, reconnecting the fragments of Figure 2, in which $op_4.prev \leftarrow op_0$, $op_1.prev \leftarrow op_3$, $op_7.prev \leftarrow op_2$ and $op_5.prev \leftarrow op_7$. Although this list contains the Nodes of all operations from tail to head, it violates linearizability because op_3 is ordered after op_2 although it follows it in real-time order. By using the two vector timestamps, our algorithm is able to order the fragments so that linearizability is maintained.

Another case that may arise is a set of fragments that consists of a single complete list with one or more singleNodes, which results from a crash that occurs when none of the pending operations completed their primitive swap operations. As we prove, in this case it is safe to put all these singleNodes (in any order) at the end of the list.

3.1 Detailed Description of the Algorithm

Data structure definitions and the pseudo-code are presented by Algorithm 1. Text in blue is for the independent failures algorithm and should be disregarded for now. We first describe key data structures and shared variables.

Each SWAP operation is represented by a single *Node* structure. *Node.val* stores the operand of the SWAP operation. *Node.seq* stores the sequence number of the current process's SWAP operation. *Node.prev* is a pointer to the *Node* structure representing the previous SWAP operation. Consequently, *Node.prev.val* stores the value that must be returned by the SWAP operation represented by *Node*. Each *Node* structure also stores two timestamp

16: **procedure** GRECOVER() ▷ Global SWAP recovery procedure
17: $V \leftarrow \emptyset$, $E \leftarrow \emptyset$
18: **for** i from 0 to n **do**
19: $currNode \leftarrow Nodes[i]$
20: **while** $currNode \neq null$ **do**
21: $V \leftarrow V \cup \{currNode\}$
22: **if** $currNode.prev \neq null$ **then**
23: $V \leftarrow V \cup \{currNode.prev\}$
24: $E \leftarrow E \cup \{(currNode, currNode.prev)\}$
25: $currNode \leftarrow currNode.prevExecution$
26: $TAILNODE \leftarrow new(Node)$
27: ▷ Add graph node representing the list's tail and graph edge pointing to the *tail* Node
28: $V \leftarrow V \cup \{TAILNODE\}$
29: $E \leftarrow E \cup \{(TAILNODE, tail)\}$
30: Compute set $Paths$ of maximal paths in graph $\mathcal{G} = (V, E)$
31: $MiddlePaths \leftarrow \emptyset$
32: $SingleNodes \leftarrow \emptyset$
33: **for** $path \in Paths$ **do**
34: **if** $len(path) == 1$ **then**
35: $SingleNodes \leftarrow SingleNodes \cup \{start(path)\}$
36: Remove $path$ from $Paths$
37: **for** $path \in Paths$ **do**
38: **if** $TAILNODE \in path$ and $Nodes[0] \in path$ **then**
▷ There is a single full path from tail to head
39: For every Node in $SingleNodes$ re-execute SWAP from Line 10
40: **return**
41: **else if** $TAILNODE \in path$ **then**
42: $TailPath \leftarrow path$
43: **else if** $Nodes[0] \in path$ **then**
44: $HeadPath \leftarrow path$
45: **else**
46: $MiddlePaths \leftarrow MiddlePaths \cup \{path\}$
47: **for** $curPath \in sort(MiddlePaths \cup SingleNodes$ in non-increasing $\succ$ order) **do**
48: $end(TailPath).prev \leftarrow start(curPath)$
49: update $TailPath$ to include added path
50: $end(TailPath).prev \leftarrow start(HeadPath)$
51: **procedure** RECOVER(val) ▷ Individual SWAP recovery procedure for process i
52: $myNode \leftarrow Nodes[i]$
53: **if** $myNode == null$ or $myNode.seq < SEQ_i$ **then**
54: **return** SWAP(val)
55: **else**
56: $Res_i \leftarrow myNode.prev.val$
57: **return** Res_i

vectors of size n, *Node.startVts* and *Node.endVts*. Lastly, *Node.prevExecution* is a pointer to the *Node* structure representing the previous SWAP operation by the same process (if there is one).

Nodes is an array of $n + 1$ pointers to *Node* structures. *Nodes*[0] points to the *headNode* sentinel node. For each process $i \in \{1, \ldots, n\}$, *Nodes*[i] points to the beginning of a list of *Node* structures, induced by *prevExecution* pointers, which represent the SWAP operations of process i. This array is used for recording all *Node* structures created throughout the execution.

VTS is an array of length n that serves as a global vector timestamp. Entry i counts the number of operations performed by process i. *tail* is a pointer to a *Node* structure. Res_i is a per-process non-volatile variable to which an operation's response is persisted before it returns. The algorithm maintains a linked list of *Node* structures representing the order of SWAP operations and tail points to the last *Node* structure in the list.

The following order relation between fragments is used by the global recovery procedure.

▶ **Definition 4.** *Given two fragments A and B, we denote $A \succ B$ if there are nodes $n_A \in A$ and $n_B \in B$ such that $n_A.startVts > n_B.endVts$. If neither $A \succ B$ nor $B \succ A$ holds, we say that A and B are $\succ$-equal.*

A SWAP operation first creates a *Node* structure and initializes it (Lines 2–3). It then increments its entry of the VTS, collects VTS, and writes the resulting vector timestamp to the *startVTS* field of its node (Lines 4–5). In Lines 6–7, the node representing the current operation is linked to the list of the previous operations executed by this process. Then, the process *announces the operation* by writing a pointer to its node to its entry of the *Nodes* array (Line 9). Next, the procedure invokes an atomic swap operation to read a node pointer from *tail* and swap it with a pointer to the node representing the current operation (Line 10). Then, the previous *tail* value is persisted to the *prev* field of the operation's *Node* structure (Line 11), thus adding this operation to the fragment of its predecessor operation. If a process executes Line 10 but the system crashes before it executes Line 11, a new fragment will result that cannot be reached from *tail* using *prev* pointers. These fragments are reconnected by the global recovery procedure GRECOVER. The operation terminates by performing a second collect of VTS, writing it to the *endVts* field, and returning the previous value stored at *prev.val* after writing it to its *Res* variable (Lines 12, 14–15).

The GRECOVER procedure (Lines 16–50) of the SWAP operation is performed by the system upon recovery. As we've mentioned before, its task is to reconnect fragments caused by a system-wide crash by creating a total order between SWAP operations that maintains linearizability. When it terminates, all the SWAP operations that were previously announced (in Line 9) are ordered in a single fragment that includes the *headNode* and the node pointed to by *tail*. As we prove, the order of operations induced by this fragment is a linearization of the execution.

After the completion of GRECOVER, the system resurrects all the processes whose SWAP operations crashed, for executing the individual recovery procedure (Lines 51–57). It first checks if the sequence number of the process' last announced operation (found in the *Nodes* array) equals SEQ_i. If it does, the value of *Nodes*[i]'s *prev* field is written to Res_i and returned (Lines 56–57). Otherwise, process i's operation crashed before it was announced and so SWAP is re-executed (Line 54).

Pending SWAP operations that updated their *prev* pointers before the crash can simply return the value stored in *prev.val*. SWAP operations that did not update their *prev* pointer in Line 11 before a crash are of two types: Those that executed Line 10 before the crash and those that did not. SWAP operations of the latter type are simpler to deal with since they did not change the *tail* pointer and can therefore be re-executed. Correctly ordering SWAP operations of the first type (i.e. those that executed Line 10 but did not execute Line 11) is more challenging, since their primitive swap operation changed *tail*'s value but its return value was lost.

We proceed to describe the GRECOVER procedure. It starts by constructing a directed graph $\mathcal{G}$ whose nodes correspond to Node structures and whose edges correspond to *prev* pointers (Lines 17–29). The construction is done by traversing (non-null) *prev* and *prevExecution* fields starting from each entry of the Nodes array (Lines 17–25). After the traversal

ends, a special TAILNODE node and an edge directed from it to the node pointed at by *tail* are added to the graph for simplifying the handling of the tail fragment (Lines 26–29). A set *Paths* of maximal directed paths in $\mathcal{G}$ is computed in Line 30. $\mathcal{G}$ is cycle-free, because each SWAP operation performs Line 10 at most once and, if it does, receives in response a pointer to an operation that performed Line 10 before it. Thus, the set *Paths* is well-defined. Each element of *Paths* represents a fragment.

Next, all singleNodes (if any) are removed from *Paths* and inserted into a separate *SingleNodes* set (Lines 32–36). If *Paths* has a fragment that contains both TAILNODE and the *headNode* sentinel node, then, as we prove, there are no middle fragments. In this case, each of the operations that correspond to *SingleNodes* is executed, in turn, starting from Line 10 and their nodes are thus appended to the end of this full path. Then GRECOVER returns (Lines 38–40). Otherwise, the fragments in *Paths* are categorized to a single *HeadPath* fragment, a single *TailPath* fragment, and a *MiddlePaths* set that contains all other paths, which are middle fragments (Lines 37–46).

Next, all fragments other than *HeadPath* and *TailPath* are sorted in non-increasing $\succ$ order (see Definition 4) and are appended, one after the other, to the end of *TailPath* by updating appropriate *prev* fields (Lines 47–49). Sorting is done by comparing *startVts* and *endVts* fields as specified by Definition 4. The construction of the full order is concluded by appending the *HeadPath* to the end of the *TailPath* (Line 50).

The execution of GRECOVER, as well as that of the individual recovery procedure, can incur one or more crashes. In this case, the recovery process is re-executed upon recovery from each crash. As we prove, when it completes, operations are ordered correctly.

3.2 Proof of Correctness

We say that the list $l = (a, a_1 = a.prev, a_2 = a_1.prev..., a_m = a_{m-1}.prev, b = a_m.prev)$, for $m \geq 0$, of length $m + 2$, is *induced by prev pointers*. We also say that l *starts at* a and l *ends at* b. We define the notion of a list *induced by prevExecution pointers* similarly.

The following lemma states that in GRECOVER at Line 30, all Node structures ever announced by Line 9 are in V, and all *prev* pointers from Node u to Node v that were set in Line 11 are represented by an edge $(u, v) \in E$.

▶ **Lemma 5.** *At Line 30,* $\{v | v$ *was announced in Line 9}* $\subseteq V \wedge \{(u, v) | u.prev = v\} \subseteq E$ *holds.*

Proof. First note that every Node structure announced at Line 9 is inserted into the Nodes array. Also, notice that each entry in the Nodes array is written to by a single process.

In addition, every time a Node x is overwritten by a new Node y in Nodes[p] by process p, $y.prevExecution$ equals x. It follows that all of the Node structures created by p can be reached by going over the list induced by *prevExecution* pointers starting from $Nodes[p]$, for all $p \in \{1 \dots n\}$. Also notice that Nodes[0] is a special case of a list of length 1, representing the initial state.

In the for loop at Line 18, the recovery process goes over all process numbers from 1 to n and also over Nodes[0]. For each process, it goes over the list induced by its *prevExecution* pointers, so that *currNode* gets assigned every Node ever announced in Line 9 as well as Nodes[0]. Consequently, $\{v | v$ was announced in Line 9$\} \subseteq V$ holds in Line 30. In addition, by going over every announced Node and adding the $TAILNODE$ link, we ensure that for every pair of Node structures u and v, if $u.prev = v$, there is an edge $(u, v) \in E$ hence $\{(u, v) | u.prev = v\} \subseteq E$. ◀

The next lemma states that every Node has at most a single *prev* or *tail* pointer pointing at it.

▶ **Lemma 6.** *For every Node u, either tail* $= u$ *and* $|\{v|v.prev = u\}| = 0$ *or* $|\{v|v.prev = u\}| \leq 1$.

Proof. We consider all code lines that assign *prev* pointers. First, notice that the lemma holds in the initial state since only *Nodes*[0] exists and is pointed only by the *tail* pointer.

When a *prev* pointer is assigned at Line 11, it is done using an atomic primitive swap operation, meaning that only a single SWAP operation can read the specific *prev* value that was previously pointed by *tail*.

When a *prev* pointer is assigned during GRECOVER it is done in either Line 48 or 50. In both cases, it is assigned with a Node that starts a path in the Graph(V, E) and by Lemma 5 all Nodes created and *prev* pointers are represented in the graph. It follows that there is no other *prev* pointer pointing to the Node assigned, because it is the start of a maximal path in the graph (V, E). ◀

We now show that the set of paths *Paths* computed in Line 30 is node-disjoint.

▶ **Lemma 7.** *Let P and J be two different paths that exist simultaneously in the global recovery process, and consider a Node* $i \in P$*, then* $i \notin J$.

Proof. From Lemmas 5 and 6, every Node $v \in V$ has at most one incoming edge, meaning there is at most one Node $u \in V$ such that $(u, v) \in E$. In addition, every edge $(u, v) \in E$ signifies that the node representing u has its *prev* pointer pointing to the node representing v. Thus, for every node $u \in V$ there is at most one $v \in V$ such that $(u, v) \in E$. Assume towards a contradiction that there exist two maximal paths P,$J \in Paths$, $P \neq J$ such that $i \in P$ and $i \in J$, then either there are nodes $u \in P$, $j \in J$ s.t $u \neq j$ and $(i, u), (i, j) \in E$, or there are nodes $u \in P$, $j \in J$ s.t $u \neq j$ and $(u, i), (j, i) \in E$. The first case yields a contradiction because node i has two *prev* pointers, and the second case means node i has two prev pointers pointing at it, yielding a contradiction to Lemma 6. ◀

The next lemma shows that after a successful global recovery, the Node list starting at tail is complete and holds all announced Node structures.

▶ **Lemma 8.** *At the end of a crash-free execution of the global GRECOVER procedure, there is a single list induced by prev pointers starting from tail and ending in* $Nodes[0]$ *such that all Node structures announced at Line 9 are in it.*

Proof. First, assume that the condition in Line 38 holds. In this case, there is a maximal *path* $P \in Paths$ that includes both TAILNODE and *Nodes*[0]. Assume towards a contradiction that the lemma does not hold. It follows that there is a Node i that isn't in *path* P. There are two sub-cases to consider. Either there is another *path* J in $Paths$, or there isn't. If the latter sub-case holds, then i must be in $SingleNodes$, hence SWAP will be re-executed starting from Line 10 on its behalf (in Line 39), and because the execution is crash-free, i will be inserted to the list induced by *prev* pointers starting at *tail*, and this list will end in $Nodes[0]$ according to the code. The former sub-case is that $i \in J$ and J is a *middle fragment*, hence it is of length at least two. Let x be the last Node in J and let Op be the operation represented by x. Op must have executed Line 10 before the crash (otherwise no *prev* pointer could point at x) but did not execute Line 11 before the crash (since x is the last node in J). Immediately after Op executed Line 10, *tail* pointed to x. Since Op did not execute Line 11, this contradicts the existence of a path in $Paths$ starting from *tail* and ending in $Nodes[0]$.

Otherwise, the condition in Line 38 does not hold. In this case, it follows from Lines 37–50 that immediately after Line 50 is executed by Op, TailPath is a list that starts from TAILNODE, ends with $Nodes[0]$, and contains all the Nodes in V. Thus by Lemma 5, this list contains all announced Nodes and the lemma holds. ◀

The following lemma ensures that the $\prec$ order can only hold in one direction for any two distinct Paths.

▶ **Lemma 9.** *Let A and B be two paths that exist simultaneously in the global recovery procedure. Then $A \prec B \implies B \not\prec A$.*

Proof. By Definition 4, $A \prec B$ implies that there is a *Node* $x \in A$ and a *Node* $y \in B$ such that $x.endVts < y.startVts$. From Lemma 7, both fragments are disjoint. A new fragment is created only when a process executes Line 10 but does not execute Line 11. This implies that all operations on fragment A that completed Line 10 had done so before any of the operations on fragment B performed Line 10, or vice versa. Since $x.endVts < y.startVts$ and $x.endVts$ is only written after executing Line 10, and since $y.startVts$ is only set before executing Line 10 and after increasing VTS, it follows that all the operations of fragment A executed Line 10 before any the operations of fragment B have executed this line.

If we also have $B \prec A$, then by Definition 4, there is a *Node* $a \in A$, and a *Node* $b \in B$, such that $b.endVts < a.startVts$. It follows that the operation represented by b executed Line 10 before the operation represented by a, which is a contradiction. ◀

The following theorem proves the correctness of our algorithm for the system-wide failures model.

▶ **Theorem 10.** *Algorithm 1 implements a recoverable NRL SWAP in the system-wide failures model using only read, write and primitive swap operations. Its SWAP operations are wait-free and it is strictly recoverable.*

Proof. Clearly from the code, SWAP operations are wait-free since they have no loops, so they terminate in crash-free executions. In addition, SWAP is strictly recoverable as its response is stored in a per-process non-volatile variable Res_i before it returns. We also observe that the algorithm is linearizable in crash-free executions: An operation Op is linearized in Line 10 when it swaps a pointer to its Node to *tail*. The next operation to execute Line 10 after Op (if any) is guaranteed to return Op's input as its response in line 15. If Op is the first SWAP operation to perform Line 10 then, from the initialization of *headNode*, it returns the object's initial value $\perp$.

Next, we consider executions that incur crashes and prove the following property: Upon completion of the GRECOVER procedure, the list $\mathcal{L}$ induced by *prev* pointers, starting from *tail*, contains, exactly once, the Node structure of every SWAP operation announced (in Line 9) in the course of the execution and ends at *headNode*. Moreover, the order induced by $\mathcal{L}$ respects operations' real-time order.

Let $N_1, N_2 = N_1.prev, N_3 = N_2.prev ... N_l = N_{l-1}.prev$ such that $N_1 = Tail$ and $N_l = Nodes[0]$ denote the Node structures in the list induced by *prev* pointers beginning at *tail*. In the initial state, $Tail = N_1 = N_l = Nodes[0]$. For a node x, denote by *proc(x)* the process that created and announced node x and let $H = (SWAP_{proc(N_{l-1})}(N_{l-1}.val), SWAP_{proc(N_{l-2})}(N_{l-2}.val)$ ... $SWAP_{proc(N_1)}(N_1.val))$, where $N_1, \ldots, N_l$ are the nodes in the single fragment that exists immediately after the GRECOVER procedure. Let α denote the execution that ends when GRECOVER completes. From Lemma 8, H is a sequential history that contains all the SWAP operations that were announced in α. Moreover, for any extension β of α in which all these SWAP operations return following the execution of their

individual RECOVER procedures, they return the same values in β and in H. Operations that weren't announced in α but whose individual RECOVER procedures were executed in β, re-execute SWAP(val) (Line 54) and are therefore linearized when they execute Line 10.

It remains to show that for any two announced SWAP operations, SWAP_1 and SWAP_2, if SWAP_1 terminates in α before SWAP_2 starts, then SWAP_1 precedes SWAP_2 in H. Let Node_1, Node_2 be the Nodes created by SWAP_1 and SWAP_2, respectively. Assume towards a contradiction that SWAP_2 precedes SWAP_1 in H. Then there is a path induced by *prev* pointers from Node_1 to Node_2 when GRECOVER terminates. There are two cases to consider. The first case is that all the *prev* pointers of the path between Node_1 and Node_2 were assigned by SWAP operations and not during recovery. This implies that between the time when SWAP_1 executed Line 10 and the time when SWAP_2 executed Line 10, no process performed Line 10 without performing Line 11, otherwise Node_1 and Node_2 would have been on separate fragments just before recovery. Consequently, Node_1 and Node_2 are on the same fragment and Node_1 precedes Node_2 before a crash. It follows that Node_1 precedes Node_2 also in the single fragment that exists when GRECOVER terminates, hence, SWAP_1 precedes SWAP_2 in H, which is a contradiction.

The second case is that the path induced by prev pointers from Node_1 to Node_2 was formed during recovery. There are two sub-cases to consider. The first is that during recovery, one of the Nodes is in *singleNodes* while the other is in a full path from TAILNODE to *Nodes*[0] (thus the condition in Line 38 is true). In this case, since operations that create Nodes in *singleNodes* did not complete, it must be that Node_2 is in *singleNodes* and Node_1 is on the full path. From Line 39, SWAP_2 will be re-executed and so Node_2 will be placed before Node_1 in the list induced by *prev* pointers starting from *tail* at the end of GRECOVER, hence, SWAP_1 precedes SWAP_2 in H, which is a contradiction.

The second sub-case is when the condition of Line 38 is not satisfied. This implies that just before crashing, Node_1 and Node_2 were on different fragments. Let A and B respectively denote the paths representing the fragments on which Node_1 and Node_2 were just before the crash. Since $\text{Node}_1.endVts < \text{Node}_2.startVts$ must hold, from Definition 4, $A \prec B$ holds. Consequently, from Lemma 9 $B \not\prec A$, therefore immediately after GRECOVER terminates there is a path from Node_2 to Node_1, hence, SWAP_1 precedes SWAP_2 in H, which is a contradiction. ◀

4 Detectable Swap Algorithm for the Independent Failures Model

In the independent failures model, each process may crash and recover independently of other processes. A recoverable algorithm for this model must therefore allow one or more processes to execute RECOVER concurrently, while other processes may concurrently execute their SWAP operations. In order to handle this concurrency correctly, we introduce two key changes to Algorithm 2. First, the RECOVER procedure now synchronizes concurrent invocations by using a starvation-free RME lock, implemented from reads and writes only, such as that presented by [20]. This serializes the execution of the recovery code. The goal of the second change is to allow the recovery code to wait for a concurrent SWAP operation *Op* to either complete or crash. Only once this happens, can the recovery code add the Node representing *Op* to graph *G*.

The few additions done in the pseudo-code of SWAP are presented in blue font in Algorithm 1. These consist of adding an *inWork* field (initialized to 0) to the *Node* structure, setting it (in Line 8) just before the SWAP operation is announced in Line 9, and resetting it (in Line 13) immediately after the *endVTS* field is updated in Line 12.

The pseudo-code of the RECOVER procedure for the independent failures model is presented by Algorithm 2. RECOVER first checks whether the *Node* of the crashed operation was announced (Line 3). If it wasn't, it re-executes SWAP(val) (Line 4). Otherwise, it signals that it is performing RECOVER by writing 2 to the *inWork* field of its Node and then attempts to acquire *mutex* (Lines 5–6). Next, it checks if the operation already has a value to return and if so, persists this value and returns it (Lines 7–8, 40–43).

Algorithm 2 Recoverable detectable SWAP, for the independent failures model.

```
1: procedure RECOVER(val)                                   ▷ executed by process i
2:     myNode ← Nodes[i]
3:     if myNode == null or myNode.seq < SEQ_i then
4:         return SWAP(val)
5:     myNode.inWork ← 2
6:     mutex.lock()
7:     if myNode.prev ≠ null then
8:         GoTo Line 40
9:     V_1, E_1 ← gatherGraph()
10:    tailNode ← tail
11:    await(tailNode.inWork ∈ {0, 2})
12:    V_2, E_2 ← gatherGraph()
13:    V_2 ← V_2 ∪ {TAILNODE, tailNode}
14:    E_2 ← E_2 ∪ {(TAILNODE, tailNode)}
15:    V ← V_1 ∪ V_2
16:    E ← E_1 ∪ E_2
17:    Compute set Paths of maximal paths in graph G = (V, E)
18:    for path ∈ Paths do
19:        if myNode ∈ path then
20:            myPath ← path
21:    if len(myPath) == 1 then
22:        re-execute SWAP from Line 10 for myNode
23:        mutex.release()
24:        Res_i ← myNode.prev.val
25:        return Res_i
26:    MiddlePaths ← ∅                                      ▷ May include SingleNodes
27:    for path ∈ Paths do
28:        if TAILNODE ∈ path then
29:            TailPath ← path
30:        else if Nodes[0] ∈ path then
31:            HeadPath ← path
32:        else
33:            MiddlePaths ← MiddlePaths ∪ {path}
34:    ordPaths ← sort(MiddlePaths) in non-increasing ≻ order
35:    candidate ←first path C ∈ ordPaths after myPath s.t. start(C) ∈ V_1 or null if no such C
36:    if candidate ≠ null then
37:        myNode.prev ← start(candidate)
38:    else
39:        myNode.prev ← start(HeadPath)
40:    myNode.endVts ← collect(VTS)
41:    mutex.release()
42:    Res_i ← myNode.prev.val
43:    return Res_i
```

Algorithm 3 The gatherGraph procedure.

```
1: procedure GATHERGRAPH()                                   ▷ Used by Algorithm 2
2:     V ← ∅
3:     E ← ∅
4:     for j from 0 to n do
5:         currNode ← Nodes[j]
6:         while currNode ≠ null do
7:             await(currNode.inWork ∈ {0, 2})
8:             V ← V ∪ {currNode}
9:             if currNode.prev ≠ null then
10:                V ← V ∪ {currNode.prev}
11:                E ← E ∪ {(currNode, currNode.prev)}
12:            currNode ← currNode.prevExecution
```

Lines 9–17 construct the graph $\mathcal{G}$. Unlike the system-wide failure construction, we go over all Nodes *twice*, thus constructing two sets of Nodes, V_1 and V_2. In addition, candidate paths chosen from $MiddlePaths$ are only chosen if their fragment starts from V_1 (Line 35). This is done because, after a single traversal that constructs V_1, there might be a Node in V_1 that is the start of a fragment that may be pointed by some $Node\ x \notin V_1$. As we prove, a second traversal ensures that the problem cannot occur for a graph constructed based on $V = V_1 \cup V_2$. During each traversal of *Nodes*, the algorithm waits for each *Node* v's *inWork* field to be 0 or 2 before adding it to V (Line 7 of *gatherGraph*). This ensures that the operation Op that created v isn't concurrently executing its critical part of SWAP and, therefore, v cannot change after being added to V.

The rest of the procedure is similar to that of GRECOVER in Algorithm 1. All maximal paths in $\mathcal{G}$ are calculated and classified to *TailPath*, *HeadPath* and *MiddlePaths*. In the end a single candidate path either from the sorted *MiddlePaths* or the *HeadPath* is selected to be linked to *myNode* (Lines 34–39). Note that as we ensure only for Nodes in V_1 that any Node pointing at them is in V, only such Nodes are considered as candidates (Line 35). As we prove, this ensures linearizability. The traversals that construct V_1 and V_2 are implemented by the helper function *gatherGraph*, whose pseudo-code is presented by Algorithm 3. The proof of Theorem 11 appears in the full version of this paper.

▶ **Theorem 11.** *Algorithm 2 implements a recoverable NRL SWAP in the independent failures model using only read, write and primitive swap operations and satisfies NRL. Its SWAP operations are wait-free and it is strictly recoverable.*

5 Impossibility of lock-freedom for the independent failures model

In this section, we prove a theorem establishing the impossibility of implementing lock-free algorithms for a wide variety of recoverable objects under the independent failures model. This generalizes previous results [2, 37] to a wider family of operations and implementations. In particular, it applies to any self-implementation of swap under the independent failures model, showing that our usage of a mutual exclusion lock in Algorithm 2 is essential.

We start by defining the notion of a *distinguishable operation*. An operation M is distinguishable, if there exists a history α_{base} in *spec* and two invocations M_p and M_q of M, such that the return values of the invocations allows the system to distinguish which operation is applied right after α_{base}. Formally:

▶ **Definition 12** (Distinguishable operation). *Operation $M : VAL \to RET$ is* distinguishable *if there exists a history α_{base} and values $x, y \in VAL$, $z \in RET$, such that if $M(x)$ and $M(y)$ are applied sequentially right after α_{base}, the first (and only the first) invocation of M to complete returns z.*

Assume a swap object with value 0 and two SWAP operations. If SWAP(1) and SWAP(2) are applied sequentially, only the first operation applied will return 0. This shows that SWAP is a distinguishable operation. Similarly, it's easy to show that pop and deque operations of the stack and queue data structures, as well as *fetch&add* and *test&set*, are also distinguishable operations.

Our impossibility result applies to implementations of distinguishable operations that use only read, write, and a set of *interfering functions*, defined as follows:

▶ **Definition 13** (Interfering functions [22]). *Let F be a set of primitive functions indexed by an arbitrary set K. Define F to be a set of* interfering functions *if for all i and j in K, for any object O that supports f_i and f_j, and for any state S of O, either*

1. *f_i* **and** *f_j* **commute***: The application of f_i to O in state S by process p followed by the application of f_j to O by process q leaves O (but not necessarily the local state of each process) in the same state as the application of f_j to O in state S by process q followed by the application of f_i to O by process p; or*
2. *f_j* **overwrites** *f_i: The application of f_i to O in state S by process p followed by the application of f_j to O by process q leaves O (but not necessarily the local state of each process) in the same state as the application of f_j to O in state S by q alone.*

A *configuration* C consists of the states of all processes and the values of all shared base objects. Sometimes we use the notions of a configuration and a history interchangeably. For example, if a finite history H leads to a configuration C we may use H for representing C when H is clear from the context. Two configurations C_1 and C_2 are *indistinguishable* to a set of processes P, denoted $C_1 \overset{P}{\sim} C_2$, if every process in P has the same state in C_1 and C_2, and every shared object holds the same value in C_1 and C_2.

Given a configuration C reached after a history α_{base}, distinguishable operation M, and a process $r \in \{p, q\}$, we say that C is *r-valent* if there is an execution starting from C in which the return value of M or M.RECOVER by r is z (where α_{base}, M and z are as in Definition 12). C is *bivalent* if it is both p-valent and q-valent, for $p \neq q$. C is *p-univalent* if it is p-valent and not q-valent, and symmetrically for q-univalent. C is *univalent* if it is either p-univalent or q-univalent. Let C be a bivalent configuration and s be a step. If $C \circ s$ is univalent, we say that s is a *critical step*. We generalize the proofs of [2, 37] to prove the following theorem by using valency arguments [15, 18].

▶ **Theorem 14.** *Let M be a distinguishable operation. There is no recoverable implementation I of M from read, write and a set of $K \geq 1$ of interfering primitive operations $f_1 \dots f_K$ in the independent failures model, such that both M and M.RECOVER are lock-free.*

Proof. Assume towards a contradiction that such a lock-free implementation exists. Assume that process p invokes M with value x and process q invokes M with value y, for x, y and z as in Definition 12.

To prove Theorem 14, we construct an execution in which each process performs an infinite number of steps and q neither crashes nor completes its operation.

Configuration C_0, reached after execution α_{base}, is bivalent because a solo execution of either p or q from C_0 returns z. Following a standard valency argument and since we assume that M is lock-free, there is a crash-free execution starting from C_0 that leads to a bivalent

configuration C_1, in which both p and q are about to execute a critical step. It must be that one step leads to a p-univalent configuration while the other leads to a q-univalent configuration. We can prove:

▷ Claim 15. The critical steps of p and q apply (possibly the same) primitives f_i and f_j, respectively, to the same base object.

Proof. Consider all possible steps: read, write, crash and $f_1 \dots f_K$. Assume s_p and s_q are critical steps by process p and q respectively, such that $C_1 \circ s_p$ is p-univalent while $C_1 \circ s_q$ is q-univalent.

- Steps s_p and s_q access distinct registers. In this case, these configurations are indistinguishable to p and q, that is, $C \circ s_p \circ s_q \overset{p,q}{\sim} C \circ s_q \circ s_p$
- Step s_q is a crash step then $C \circ s_p \circ s_q \overset{p}{\sim} C \circ s_q \circ s_p$
- Steps s_p and s_q read the same register. Also in this case $C \circ s_p \circ s_q \overset{p,q}{\sim} C \circ s_q \circ s_p$
- Step s_p writes to some register r step and s_q reads r. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Step s_p applies f_i $1 \leq i \leq K$ and step s_q reads r. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Steps s_p and s_q write to the same register. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Step s_p applies f_i, $1 \leq i \leq K$, step s_q writes to the same register. In this case, $C \circ s_q \overset{q}{\sim} C \circ s_p \circ s_q$ holds.
- Step s_p applies f_i, $1 \leq i \leq K$, step s_q applies f_j, $1 \leq j \leq K$ each to a different base object O, In this case $C \circ s_p \circ s_q \overset{p,q}{\sim} C \circ s_q \circ s_p$ holds.

In each of the above cases, the configurations are indistinguishable to at least one process, and therefore, must have the same valencies. Therefore, it must be that p and q apply f_i and f_j respectively to the same base object. ◁

Assume, without loss of generality, that $C_1 \circ p$ is p-univalent while $C_1 \circ q$ is q-univalent. We consider two cases:

Case 1: f_i and f_j commute. Consider executions C_2, C_3 where $C_2 = C_1 \circ p \circ q \circ CRASH_p$ and $C_3 = C_1 \circ q \circ p \circ CRASH_p$. Configurations C_2 and C_3 are reached after p and q each take a step (in different orders) in which they apply their operations to the same base object O and then p crashes.

A solo execution of M.RECOVER by p from both C_2 and C_3 must complete since I is lock-free. Furthermore, $C_2 \overset{p}{\sim} C_3$ holds, because p's response from the primitive f_i is lost, while the value of O is the same in both configurations since f_i and f_j commute. Consequently, an execution of M.RECOVER by p from both C_2 and C_3 must return the same value. Let v denote this value.

Assume first that $v = z$ and thus C_3 is p-valent. Configuration $C_1 \circ q \circ p$ is q-univalent, while $C_3 = C_1 \circ q \circ p \circ CRASH_p$ is p-valent. However, $C_1 \circ q \circ p \overset{q}{\sim} C_3$ holds because q is unaware of p's crash. Consequently, a solo execution of q from C_3 must return z, that is, C_3 is also q-valent. This proves that C_3 is bivalent.

Assume then that $v \neq z$. We now show that, in this case, C_2 is bivalent. Indeed, from this assumption, C_2 is q-valent, because a solo execution of q after p completes (and returns $v \neq z$) must return z since, from Definition 12, exactly one of these two operations must return z. However, configuration $C_1 \circ p \circ q$ is p-univalent, while $C_2 = C_1 \circ p \circ q \circ CRASH_p \overset{q}{\sim} C_1 \circ p \circ q$, therefore a solo execution of q from C_2 must return x s.t. $x \neq z$. Thus, C_2 is bivalent.

Case 2: f_j "overwrites" f_i. Consider executions C_2, C_3 where $C_2 = C_1 \circ p \circ q \circ CRASH_p$ and $C_3 = C_1 \circ q \circ CRASH_p$. A solo execution of M.RECOVER by p from both C_2 and C_3 must complete since I is lock-free. Furthermore, $C_2 \overset{p}{\sim} C_3$ because p's response from the primitive f_i is lost, while the value of the base object f_i and f_j are applied to is the same in both configurations since f_j "overwrites" f_i. Therefore, an execution of M.RECOVER by p from both C_2 and C_3 returns the same value. Let v denote this value.

Assume $v = z$ and thus C_3 is p-valent. $C_1 \circ q$ is q-univalent, while $C_3 = C_1 \circ q \circ CRASH_p$ is p-valent. $C_1 \circ q \overset{q}{\sim} C_3$ holds because q is unaware of p's crash. Therefore, a solo execution of q from C_3 returns z, that is, C_3 is also q-valent. This proves that C_3 is bivalent.

Assume then that $v \neq z$. We show that in this case C_2 is bivalent. Indeed, from our assumption, C_2 is q-valent, as a solo execution of q after p completes must return z since, from Definition 12, exactly one of these two operations must return z. However configuration $C_1 \circ p \circ q$ is p-univalent and $C_2 = C_1 \circ p \circ q \circ CRASH_p \overset{q}{\sim} C_1 \circ p \circ q$, therefore a solo execution of q from C_2 must return $x \neq z$. This establishes that C_2 is bivalent.

In both cases, this shows that we can keep extending the execution obtaining an infinite execution in which neither p nor q complete their operations and q performs an infinite number of steps without crashing, contradicting the lock-freedom assumption. ◀

Golab proves for the independent failures model that any commutative/overwriting primitive with consensus number 2 drops to level 1 in the recoverable consensus hierarchy [18, Theorem 4.9]. Although the model he assumes differs from ours (e.g., there is no separation between an operation and its recovery code) and the problems for which the impossibility results hold are different as well, our proof is similar to his. It might be possible to prove Theorem 14 by using a reduction from [18, Theorem 4.9]. However, we think that our simple direct proof makes the presentation clearer and more self-contained.

6 Discussion

We present two NRL self-implementations of the swap object, one for the system-wide failures model and the other for the independent failures model. In both, SWAP operations are wait-free and the recovery code is blocking. In the system-wide failures model, this is a result of delegating the recovery to a single process, while in the independent failures model, it is due to coordination between the recovering process and the other processes. Both our algorithms support a strictly recoverable SWAP operation. We also prove the impossibility of a lock-free implementation of distinguishable operations using read-write and a set of interfering functions, in the independent failures model. In particular, this shows that with independent failures, a self-implementation of swap cannot be lock-free.

Our algorithms use $O(m * n)$ space, where m is the number of SWAP invocations in the execution. Bounding memory consumption to $O(n)$ is relatively easy if a recoverable SWAP operation by one process can wait for operations by other processes to either make progress or fail. An interesting open question is to figure out whether the space complexity of detectable swap self-implementations with wait-free operations can be reduced to $o(m)$ or if $\Omega(m)$ is inherently required. We leave this question for future work.

Finally, it is also important to explore how self-implementations, in particular of SWAP, can be used to turn non-recoverable higher-level objects into NRL implementations of the same objects.

References

1. Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.
2. Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 7–16, 2018. doi:10.1145/3212734.3212753.
3. Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 11–20, 2020. doi:10.1145/3382734.3405725.
4. Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2019. doi:10.1145/3323165.3323187.
5. Naama Ben-David, Michal Friedman, and Yuanhao Wei. Survey of persistent memory correctness conditions. *arXiv preprint*, 2022. arXiv:2208.11114.
6. Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICS.OPODIS.2015.20.
7. Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014. doi:10.1145/2660193.2660224.
8. David Yu Cheng Chan and Philipp Woelfel. Recoverable mutual exclusion with constant amortized RMR complexity from standard primitives. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 181–190, 2020. doi:10.1145/3382734.3405736.
9. Kyeongmin Cho, Seungmin Jeon, and Jeehoon Kang. Practical detectability for persistent lock-free data structures. *arXiv preprint*, 2022. arXiv:2203.07621.
10. Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 259–269, 2018. doi:10.1145/3210377.3210400.
11. Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282, 2018. doi:10.1145/3210377.3210392.
12. Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020. doi:10.1145/3342195.3387515.
13. Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference*, pages 373–386, 2018. URL: https://www.usenix.org/conference/atc18/presentation/david.
14. Panagiota Fatourou, Nikolaos D Kallimanis, and Eleftherios Kosmas. The performance power of software combining in persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 337–352, 2022. doi:10.1145/3503221.3508426.
15. 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.
16. Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 377–392, 2020. doi:10.1145/3385412.3386031.
17. Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018. doi:10.1145/3178487.3178490.

18 Wojciech Golab. The recoverable consensus hierarchy. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 281–291, 2020. doi:10.1145/3350755.3400212.

19 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 211–220, 2017. doi:10.1145/3087801.3087819.

20 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016. doi:10.1145/2933057.2933087.

21 Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 400–407. IEEE, 2004. doi:10.1109/ICDCS.2004.1281605.

22 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

23 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

24 Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 429–442, 2021. doi:10.1145/3445814.3446710.

25 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016. doi:10.1145/2872362.2872410.

26 Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing: 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings 30*, pages 313–327. Springer, 2016. doi:10.1007/978-3-662-53426-7_23.

27 Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Durable algorithms for writable ll/sc and cas with dynamic joining. *arXiv preprint*, 2023. arXiv:2302.00135.

28 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Optimal recoverable mutual exclusion using only fasas. In *Networked Systems: 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9–11, 2018, Revised Selected Papers*, pages 191–206. Springer, 2019. doi:10.1007/978-3-030-05529-5_13.

29 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic RMR on both CC and DSM. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 177–186, 2019. doi:10.1145/3293611.3331634.

30 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant RMR recoverable mutex under system-wide crashes. *arXiv preprint*, 2023. arXiv:2302.00748.

31 Prasad Jayanti and Anup Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.30.

32 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. *Computing*, 104(10):2225–2252, 2022. doi:10.1007/S00607-022-01105-1.

33 Daniel Katzan and Adam Morrison. Recoverable, abortable, and adaptive mutual exclusion with sublogarithmic RMR complexity. In *24th International Conference on Principles of Distributed Systems*, 2021. doi:10.4230/LIPICS.OPODIS.2020.15.

34 Nan Li and Wojciech Golab. Detectable sequential specifications for recoverable shared objects. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.29.

35 Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, et al. Persistent memory transactions. *arXiv preprint*, 2018. arXiv:1804.00701.

36 John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991. doi:10.1145/103727.103729.

37 Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Recoverable and detectable fetch&add. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.OPODIS.2021.29.

38 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.37.

39 Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163. IEEE, 2019. doi:10.1109/DSN.2019.00028.

40 Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. Flat-combining-based persistent data structures for non-volatile memory. In *Stabilization, Safety, and Security of Distributed Systems: 23rd International Symposium, SSS 2021, Virtual Event, November 17–20, 2021, Proceedings 23*, pages 505–509. Springer, 2021. doi:10.1007/978-3-030-91081-5_38.

41 David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, pages 1–8, 2015. doi:10.1145/2803140.2803144.

42 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019. doi:10.1145/3360554.

Silent Programmable Matter: Coating

Alfredo Navarra
Department of Mathematics and Computer Science, University of Perugia, Italy

Francesco Piselli[1]
Department of Mathematics and Computer Science, University of Perugia, Italy

Abstract

By *Programmable Matter* (PM) is usually meant a system of weak and self-organizing computational entities, called *particles*, which can be programmed via distributed algorithms to collectively achieve some global tasks. We consider the SILBOT model where particles are modeled as finite state automata, living and operating in the cells of a hexagonal grid. Particles are all identical, executing the same deterministic algorithm which is based on local observation of the surroundings, up to two hops. Particles are asynchronous, without any direct means of communication and disoriented but sharing a common handedness, i.e., *chirality* is assumed. Within such a basic model, we consider a foundational primitive for PM, that is *Coating*: a set of n particles must move so as to ensure the closed surrounding of an object occupying some connected cells of the grid. We present an optimal deterministic distributed algorithm – along with the correctness proof, that in $\Theta(n^2)$ rounds solves the Coating problem, where a round concerns the minimal time window within which each particle is activated at least once.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Concurrency; Theory of computation → Self-organization

Keywords and phrases Programmable Matter, Coating, Asynchrony, Stigmergy

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.25

Funding This work has been funded by the European Union – NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 – VITALITY – CUP J97G22000170005, and by the Italian National Group for Scientific Computation (GNCS-INdAM).

1 Introduction

In distributed computing, the research for a basic model to address different situations is certainly one of the mostly investigated issues. On the one hand there is reality, with powerful entities like agents or robots that provide many useful capabilities. On the other hand, models should be formalized with the minimal set of assumptions. In fact, the weaker a model, the wider the range of applications, as well as dependability and robustness with respect to possible disruptions increase. Consequently, also the algorithms designed for weak models turn out to cover a wider set of possible scenarios. Within this context, we investigate on the so-called Programmable Matter (PM). This is intended as a matter with the ability to change its physical properties (e.g., shape, optical properties, etc.) in a programmable way. From its origin [36], PM denotes a system of computational entities, called *particles*, that can be programmed via distributed algorithms to collectively achieve global tasks. Initially, such systems have found main interest in the field of biology and nano-technologies, see e.g., [14, 34]. However, more and more natural applications in various contexts can be imagined, including smart materials, ubiquitous computing, repairing at microscopic scale, and tools for minimally invasive surgery.

[1] Corresponding author.

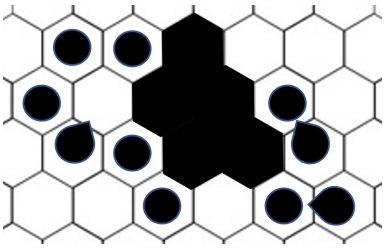


Figure 1 A configuration in SILBOT with an object represented by five connected full-black cells, seven CONTRACTED particles represented by black circles and three EXPANDED particles represented by black drop-like shapes expressing the intent to move toward a neighboring cell.

So far, one of the most investigated models for PM is certainly the geometric *Amoebot* [22, 24, 21]. The name comes from the behavior of the amoebae. Particles are modeled as finite state automata, living and operating in the cells of a hexagonal grid. They are all identical, executing the same distributed algorithm based on local observation of the surroundings, memory and exchanged messages.

Recently, in [21], an approach to homogenize the referred literature on Amoebot has appeared. One of the main intent was to enhance the model with concurrency. Along that line, one of the weakest models for PM, that includes concurrency and eliminates direct communication among particles as well as local and shared memory, is SILBOT [16, 15]. The aim has been to investigate on the minimal settings for PM under which it is possible to accomplish basic global tasks in a distributed fashion. Actually, SILBOT assumes a 2 hops distance visibility for the particles rather than just 1 hop as for Amoebot. It is worth remarking that the observation of the surrounding is the only means for the particles in SILBOT to collect information. Hence such a view must be thought to concern not only other particles in the neighborhood but also any other " visible" information within the sensing range. To that respect, particles in Amoebot are less powerful. However, the information that can be obtained by means of communications (and memory) in Amoebot may concern particles that are very far apart from each other.

In SILBOT, particles operate independently, in a fully asynchronous way, and they admit no persistent memory. Of course, an algorithm designed within such a weak context allows to build a safe, energy-efficient and fault-tolerant system when applied in real scenarios.

In SILBOT, the movement of a particle occurs from one cell to another by alternating between a CONTRACTED state (a particle uniformly occupies a central area of one cell) and an EXPANDED state (a particle occupies one cell but its disposal expresses the intent to move toward a neighboring cell), see, e.g., Figure 1.

Relative positioning among particles is their only (implicit) way of communicating, i.e., stigmergic paradigms are exploited.

Within such a weak system, fundamental tasks have been solved so far, like the *Leader Election* in [15]; the *Scattering* problem in [32], where the particles are required to reach a configuration where they are at distance at least two hops from each other; the *Line formation* in [30, 31]. One of the main open questions posed is about the solvability of other basic primitives.

1.1 Our results

In this paper, we investigate the *Coating* problem in SILBOT. Given a convex object occupying some connected cells of the grid and given a sufficiently large set of particles, the problem asks for a distributed algorithm that moves particles so as to surround the object

by occupying all the cells adjacent to it. For the ease of the discussion, we prefer to deal with convex objects. In fact, when dealing with concave objects, still particular cases must be excluded, see, e.g. the so-called *tunnels* as in [26, 20].

We provide a deterministic and distributed algorithm that optimally solves the Coating within $\Theta(n^2)$ rounds, with n being the number of particles, and a round being the minimal time window within which each particle is activated at least once. Moreover, while solving the Coating, our algorithm also resolves the *Hole Compaction*, where a hole is a connected subset of empty cells enclosed by particles and possibly the object, provided that particles are able to recognize holes. An example of hole composed of just one cell can be seen in the top-left of Figure 1, where an empty cell is surrounded by five particles (one of which is expanded) and the object. Furthermore, we prove that the designed algorithm is optimal in terms of number of rounds.

1.2 Related work

The Coating problem has been widely investigated in PM but all the approaches so far are mainly based on stronger assumptions with respect to our algorithm. In particular, the only assumption we share with such works is about chirality, i.e., particles agree on the clockwise and anti-clockwise directions.

The relevance of Coating has been deeply provided for instance in [3] where it is envisioned how PM (intended in terms of swarm of nanobots) in the future might be used to attack cancer cells, surrounding them (i.e., solving the Coating) and then destroy them so as to stop the further growth of the disease. In their approach, first the Leader Election is solved, and then the leader coordinates the other particles so as to accomplish the Coating task for a cave-shaped object.

Another approach used in [4] consists of making the particles able to calculate their distance from the object, and to set an internal value called *Own_Distance*. After each particle have calculated its *Own_Distance*, they send this information to the other particles so that everyone knows each particle's relative distance from the object. Unbounded memory is considered. Then, the particles with the biggest distance from the object send this information to the ones that have a smaller *Own_Distance* so that these ones can make space for the furthest ones and everyone can get closer to the object.

In [20], the Amoebot model is considered. The problem is solved by means of a randomized algorithm, with high probability, within $O(n)$ rounds.

A different model introduced in [26] that reminds PM is the so-called *Pairbot*. In such a work, Coating has been considered and everything revolves around the Pairbot, a single unit formed by two distinguished robots which are each other's *buddy*. It is assumed that initially the pairbots are line-formed and they all agree on the direction where to move in order to reach the object. The movement is guided by the head pairbot which starts by changing state from *short*, where the two buddies are occupying the same cell, to *long*, where the two robots occupy two adjacent cells.

3D Coating has been faced in [35] within the *3D Catom* model [33]. This is a micro-scale lattice-based modular robot that has been investigated also in the context of PM. In fact, modular robots represent a good means for implementing PM theory. They refer to robotic systems where interconnected individual (electro-mechanical) modules can recover from failures or rearrange in order to better adapt to their task-environment (see, e.g. [1, 5, 6, 13, 25, 37, 38]). Recently, the 3D environment has also been approached in [29] for a single agent that has to move tiles in order to cover the surface of an object.

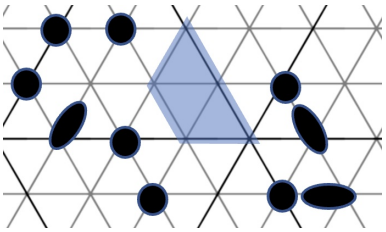


Figure 2 The same configuration of Figure 1 but with the triangular lattice representation. The object is represented by the trapezoidal shaded area; CONTRACTED particles are represented by black circles; EXPANDED particles are represented by black ellipses occupying one node and one neighboring edge.

1.3 Outline

In the next section, we review the SILBOT model in detail and we formalize the Coating problem. In Section 3, we describe and formally present our algorithm for Coating. In Section 4, the correctness proof for the proposed algorithm is presented. Finally, in Section 5, we provide concluding remarks and pose some interesting research directions.

2 SILBOT model and the Coating problem

In this section, we review the SILBOT model for PM introduced in [15], and we formalize the Coating problem.

Operating Environment. Particles operate on an infinite triangular lattice (representing the described hexagonal grid) embedded in the plane, where each node has six incident edges: nodes correspond to hexagonal cells and each edge represents a boundary shared by two cells. Each node can be occupied by at most one particle. There are n particles in the considered system and there is an object occupying a connected and convex set of nodes, see, e.g., Figure 2. From now on we will always refer to the lattice representation, hence cells are referred to as nodes.

Particles and Configurations. Each particle is an automaton with two states, CONTRACTED or EXPANDED (they do not have any other form of persistent memory). In the former state, a particle occupies a single node of the lattice while in the latter, the particle occupies one single node and one of the adjacent edges. Hence, a particle always occupies one node, at any time.

Each particle can sense its surroundings up to a distance of 2 hops, i.e., if a particle occupies a node v, then it can see the neighbors of v and the neighbors of the neighbors of v. Specifically, a particle can determine (i.e., sense) if a node is empty or occupied by a CONTRACTED particle, or occupied by an EXPANDED particle, for each node in its 2-hop visibility range.

Any positioning of CONTRACTED or EXPANDED particles that includes all n particles composing the system plus the object is referred to as a *configuration*.

Furthermore, particles have the so called *exterior awareness*, that informally is the power of detecting what is "outside" the configuration and what is "inside" (that is, *holes*).[2] Formally, a hole is a subset of empty nodes enclosed by particles and possibly by the object.

[2] It is worth remarking that the exterior awareness as well as many other useful information can be obtained in other systems by means of communications combined with memory, both absent in SILBOT.

A particle can distinguish whether a node v within its visibility range is CONT, EXP, IN, OUT or OBJ where: a CONT node is a node occupied by a CONTRACTED particle; an EXP node is a node occupied by an EXPANDED particle; an IN node is an empty node that is part of a hole; an OUT node is an exterior empty node (an empty node that is not part of any hole); an OBJ node is a node occupied by the object. From a practical point of view, the exterior awareness can be implicitly provided by a different level of light among in and out, or a different level of humidity, pressure or the presence/absence of some substance like a liquid or a gas.

Although the exterior awareness might seem a strong assumption, it is worth remarking that the sensing capability of the robots is the only means to acquire information of the surrounding as they cannot exchange messages and have no memory.

Initially, it is assumed that particles are all CONTRACTED and along with the object and possible holes, they constitute a connected set of nodes.

In what follows, we show that starting from any initial configuration, the particles endowed with the simple capabilities described above are able to achieve compaction of holes, if any, while solving Coating. Note that, during the process, the set of particles plus holes plus the object always forms a connected set of nodes.

It is worth remarking that in SILBOT, particles do not have any explicit means of communication. Thus, a particle can acquire information about its surroundings only via its limited view, by means of direct sensing, e.g., weak electromagnetic fields or radars.

Movement and States. As described above, each particle p can occupy only one node v at a time. In order to move to a neighboring node u, p expands on the edge (v, u). Thus, in the EXPANDED state, p occupies node v and edge (v, u) (the physical interpretation is that the particle is occupying one hexagonal cell and has partially entered into the adjacent one, see Figure 1). Note that node u may still be occupied by another particle p'. If p' leaves node u in the future, then p, the EXPANDED particle, will contract into node u during its next activation. There might be arbitrary but finite delays between the actions of these two particles, while the connectivity is still maintained. For example, when p' has moved to another node, edge (v, u) is still occupied by p, the originally EXPANDED particle. In this case, we say that node u is *semi-occupied*. We denote by SO the set of semi-occupied nodes. We ensure that the set of occupied and semi-occupied nodes (plus holes and the object) always induces a connected configuration during the execution of the proposed algorithm.

A very strong constraint of SILBOT, called *Commitment Property*, is that:
A particle commits itself into moving to node u by expanding in that direction, and at the next activation of the same particle, it is constrained to move to node u, if u is empty. A particle cannot revoke its expansion once committed.

Asynchrony and Rounds. The SILBOT model introduces a fine grained notion of asynchrony with possible delays between observations and movements performed by the particles. This reminds the so-called ASYNC model designed for theoretical models dealing with mobile and oblivious robots (see, e.g., [9, 7, 8, 10, 11, 17, 19, 18, 28, 27]). All operations performed by the particles are non-atomic: that is, there can be arbitrarily finite delays between the actions of sensing the surroundings, computing the next decision (e.g., expansion or contraction), executing the decision.

There are no assumptions nor restrictions on the scheduling of these events; thus any possible execution of an actual physical system can be captured by the model. This has important consequences for computability of the particle systems and requires more rigorous techniques for proving correctness of the algorithms (see, e.g. [8, 12, 11]). In particular, algorithms for this model must be inherently simple with a few rules, since this already provides an uncountable large number of possible execution sequences.

A *round* is the time within which all particles have been activated and concluded their activation time at least once. Clearly, the duration of a round is finite but unknown and may vary from time to time.

We include the well-established fairness assumption that each particle must be activated within finite time, infinitely often in any execution of the particle system. Due to the asynchronous nature of the system, it may happen that a particle decides (or is forced, in case of contraction) at time t to take an action, and that this action will actually be executed at time $t' > t$, when other particles might have changed their state; in other words, the action executed at time t' might be based on the obsolete observation of the surrounding taken at time t. The time required to accomplish an action is finite but arbitrary. Hence a round is the shortest time period during which each particle has performed an action (where an action could be the *nil* one if a CONTRACTED particle decides to remain as such, or if an EXPANDED particle finds the target node still occupied).

Orientation and Chirality. We do not make any additional assumptions about the local coordinate systems of a particle but we assume handedness, that is particles agree on *chirality*, i.e., on the clockwise and anti-clockwise directions. Various papers on Coating assume chirality (e.g., [2, 23, 24]). The removal of such an assumption for the Coating problem remains an open problem. Intuitively, in SILBOT, if two sets of particles start surrounding the object along opposite directions, a deadlocked situation might be reached where EXPANDED particles cannot reverse their direction of expansion.

Randomness. We do not assume the possibility of using randomness: particles take deterministic decisions and do not have access to random numbers. Each particle may be activated at any time independently from the others. Once activated, a particle looks at its surrounding (i.e., at its neighbors and at the neighbors of its neighbors) and, on the basis of such an observation, decides (deterministically) its next *action* as follows:

1. A CONTRACTED particle occupying node v, when activated may become EXPANDED, thus occupying node v and one of the edges leading to a neighboring node u, if the edge (v, u) is unoccupied. In this case, we say that the particle *expands* along edge (v, u) and toward node u;
2. An EXPANDED particle occupying node v and edge (v, u), when activated, always *contracts* to u (i.e., moves to u changing its state to CONTRACTED), if u is unoccupied. If u is occupied, then the particle does not change state or location. In other words, as dictated by the Commitment property, a particle in EXPANDED state is obliged to *contract* as soon as it is activated and the destination node is empty.

The activation of each particle is intended to be decided by an "adversarial" scheduler and cannot be controlled by the algorithm designer. If two CONTRACTED particles decide to expand on the same edge simultaneously, exactly one of them (arbitrarily chosen by the scheduler) succeeds. If two particles are EXPANDED along two distinct edges incident to the same node u, and both particles are activated simultaneously, exactly one of them (again, chosen arbitrarily by the scheduler) contracts to node u, while the other particle does not change its EXPANDED state.

Connectivity. An important property we preserve is that the set of particles plus the object plus the set of holes never get disconnected, apart for the aforementioned semi-occupied nodes. As in [15], we allow the particles to move asynchronously while maintaining a " relaxed" sense of connectivity with the object. We remind that when a particle in a node

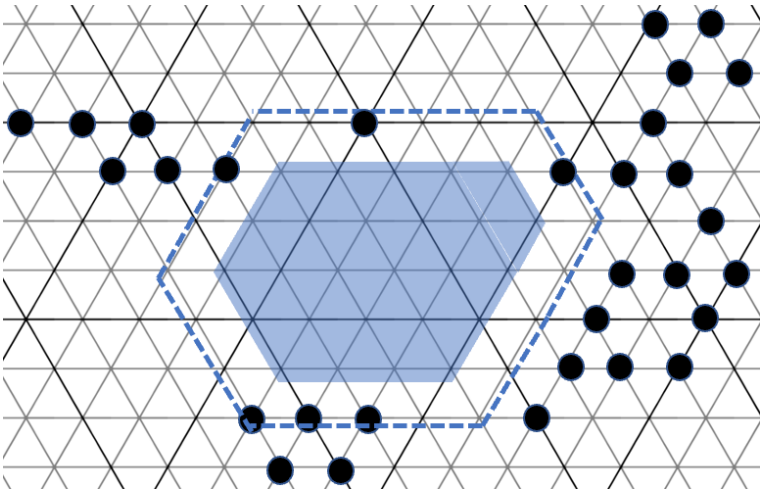


Figure 3 Example of a connected configuration where black circles represent particles; the shaded area includes all the nodes occupied by the object; the dashed line represents EP.

v is EXPANDED along edge (v, u), and u is empty, then u is considered as semi-occupied. Throughout the paper, we say that a configuration is *connected* if the set of nodes that are occupied or semi-occupied or containing the object or holes form a connected subgraph of the lattice. The algorithm we provide to solve the Coating problem always maintains a "relaxed" connected configuration while reduces possible holes and unoccupied nodes surrounding the object.

Given a triangular lattice G, a subgraph of G is *simply connected* if the envelope of its standard planar embedding (i.e. the area delimited by the subgraph) has only one exterior boundary and no interior boundaries (i.e., no holes). A configuration is said to be *simply connected* if the subgraph of lattice G induced by the nodes occupied by particles plus the object is simply connected.

Coating and Holes Compaction Problem. We assume the system is initially in a connected configuration where n particles are all CONTRACTED and sufficient to surround an object *obj*. Note that *obj* constitutes any convex subset of connected nodes.

▶ **Definition 1.** *The* External Perimeter *(EP) of obj is considered to be formed by all the nodes adjacent to obj.*

▶ Remark. Due to the above definition and to the assumed chirality, given a node $v \in EP$, the predecessor and the successor of v in EP in the clockwise (and hence also in the anti-clockwise) order are well-defined.

▶ **Definition 2** (Coating and Holes Compaction Problem (CHCP))**.** *Given an initial configuration of $n \geq |EP|$ particles, an algorithm solves* CHCP *if there exists a time t after which no expansion occurs and the following conditions hold:*

i) *the configuration is simply connected;*

ii) *the external perimeter EP is fully occupied by particles.*

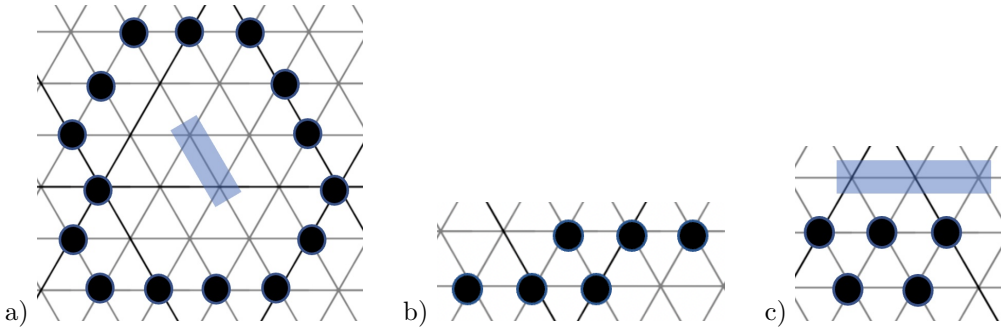


Figure 4 *a*) Possible initial instance where the object (represented by the rectangular shaded area) is completely surrounded by holes, in fact there is no particle in EP; *b*) An instance representing the *bridge problem* where the particles of degree 3 shouldn't expand, especially in opposite directions, as otherwise they may disconnect the particles; *c*) A portion of a possible configuration that requires the rule at Line 12 of Algorithm 1.

3 Algorithm CHC

In this section, we propose an algorithm for CHCP by exploiting the very basic capabilities of the particles dictated by the SILBOT model. In particular, starting from any initial connected configuration of CONTRACTED particles (not necessarily simply connected), the system eventually leads particles to surround *obj* and compact any possible hole.

The pseudocode of Algorithm CHC is reported in Algorithm 1. It is described from the point of view of a single particle. Before analyzing all details, we need the following further notation: given a particle p, we denote by $N(p)$ the set of six nodes adjacent to p, by $N(p, C)$ and $N(p, I)$ the set of CONT and IN nodes, respectively, adjacent to p, and by $N(p, IC)$ the set $N(p, I) \cup N(p, C)$.

The rationale of Algorithm CHC is to make the most external particles EXPANDED toward the internal ones, until " pushing" those adjacent to *obj*. Once particles on EP start to be pushed from the external, they can start moving along EP toward the successive position, according to the ordering provided by Definition 1, e.g., the anti-clockwise direction. In this way, the pushing from the external makes more and more particles move along EP until the Coating problem is solved. In the meanwhile, possible holes are also compacted. A slightly different behavior may occur when, in the initial configuration, *obj* is already surrounded by particles but nodes in EP are not totally occupied, i.e., EP contains some holes, see, e.g., Figure 4.a where EP contains no particles.

Following the aforementioned approach, Algorithm CHC determines two different behaviors according to whether a particle belongs to EP or not:

For particles not in EP, the algorithm allows to expand only particles p such that $G(N(p, IC))$ is connected, there are no semi-occupied nodes in $N(p)$ and $|N(p, IC)| \leq 3$. When a node in $N(p, IC)$ is an empty internal node (i.e., a hole) with $|N(p, I)| = 1$ and p is allowed to expand, then p always expands toward the internal node. In particular, if there exists a CONTRACTED particle p such that $|N(p, IC)| = |N(p, I)| = 1$ (Line 3), then p expands toward the only internal node, while if $|N(p, IC)| = |N(p, C)| = 1$, it expands toward the only CONTRACTED particle.

If $|N(p, IC)| = 2$, instead, then:

i) if there is a node $q \in N(p, I)$, p expands along edge (p, q) (Line 5);

ii) otherwise, it expands toward any CONTRACTED particle (Line 6).

It is worth remarking that holes have priority with respect to CONTRACTED particles.

Algorithm 1 Algorithm CHC (Coating and Holes Compaction).

Require: A connected configuration (including holes and the object) with node p occupied by a CONTRACTED particle.
Ensure: Coating plus Hole Compaction.
1: **if** $p \notin EP$ **then**
2: **if** $G(N(p, IC))$ is connected $\wedge$ $N(p) \cap SO = \emptyset$ **then**
3: **if** $N(p, IC) = \{q\}$ **then** Expand along (p, q);
4: **if** $N(p, IC) = \{q, r\}$ **then**
5: **if** $q \in N(p, I)$ **then** Expand along (p, q)
6: **else** Expand along (p, r);
7: **if** $N(p, IC) = \{r, q, s\}$, with q being between r and s **then**
8: **if** $\{r, q, s\} \cap N(p, I) = \{x\}$ **then** Expand along (p, x)
9: **else**
10: **if** $|N(r, C)| > 2 \wedge |N(q, C)| > 3 \ \wedge |N(s, C)| > 2$ **then** Expand along (p, q)
11: **else**
12: **if** $\{r, q\} \subseteq N(p, C) \cap EP \wedge |N(r, C)| = 2 \ \wedge |N(q, C)| > 3 \wedge |N(s, C)| > 2$ **then** Expand along (p, q);
13: **else** let r, p and q be sequential along EP;
14: **if** $N(p, C) \setminus EP = \emptyset$ **then**
15: **if** $\exists\ x$ in $N(p) \setminus EP$: x is EXPANDED toward $p \vee r$ in EP is EXPANDED toward p **then** Expand along (p, q);

If $|N(p, IC)| = 3$, then we have three cases:

i) If there is exactly one empty internal node among the three neighbors, then p expands toward it (Line 8);

ii) If the condition at Line 10 is satisfied then p expands toward the central CONTRACTED neighbor (Line 10). This is introduced in order to avoid that in a configuration similar to that depicted in Figure 4.b, particles with degree 3 expand toward opposite directions, hence potentially producing a disconnection. We refer to this situation as the *bridge problem*;

iii) If the condition at Line 12 is satisfied, then p expands toward EP (Line 12). Note that, this case captures a special occurrence as depicted in Figure 4.c. Basically, the presence of *obj* requires a special rule in order to not incur in a deadlock where no particle can expand.

For particles in EP, the algorithm allows the movement of a particle p in EP only if there is at least one particle EXPANDED toward p and all adjacent particles of p not in EP, if any, is EXPANDED. The movement is performed along nodes in EP as well (Line 15). Basically, the idea is that particles in EP tend to occupy new empty nodes of EP only when there is at least one particle which is EXPANDED toward EP and hence eventually moves there. In so doing, a particle in EP remains in EP forever. Note that $|EP|$ particles are required and sufficient to solve the Coating (and Holes Compaction) problem.

It is worth reminding that Algorithm CHC does not deal with EXPANDED particles as those are forced (under the control of the adversarial scheduler) to move toward the chosen direction as soon as possible as dictated by the Commitment property.

In the next section, we show that Algorithm CHC converges to a simply connected configuration where each node in EP is occupied within a finite number of rounds. Note that, the algorithm deals also with (not necessarily simply) connected configurations since there might occur semi-occupied nodes meanwhile. Moreover, the final configuration reached by the algorithm is either composed by all EXPANDED particles and all the nodes in EP occupied, or exactly $n = |EP|$ CONTRACTED particles occupying EP.

4 Correctness

In this section, we show the correctness of Algorithm CHC, i.e., it terminates in a finite number of rounds in a connected configuration where the object is totally surrounded and no holes appear anymore.

We show that, in a connected configuration, there always exists (at least) a particle that can expand, until the Coating (and the Hole Compaction) is solved. Moreover, during the execution of the algorithm, all the generated configurations are guaranteed to be connected (in the "relaxed" way that includes semi-occupied nodes and holes), while the number of empty nodes in EP decreases.

The proof follows an interesting technique where we model each execution of Algorithm CHC as a path in a directed graph $H = (V, E)$, where, given an object, the vertices in V correspond to any connected configuration with $n \geq |EP|$ particles, and the edges in E correspond to transitions among configurations determined by Algorithm CHC. In particular, each vertex $u \in V$ corresponds to a configuration C_v, and there is a directed edge $(v, u) \in E$ if there exists an execution of Algorithm CHC that leads from C_v to C_u, without generating in between further configurations different from C_u.

Note that, each vertex in H only encodes the positions of the particles and not their actual states. I.e., if during an execution of Algorithm CHC a particle p has already decided to expand from a configuration C_v but the expansion occurs after other actions taken by different particles, it means that from C_v there is an edge leading to the configuration C_u where p is EXPANDED and all other particles involved in actions have already computed their actions. However, graph H also contains an edge from C_v to another configuration C_w that differs from C_v only by the expansion of particle p, since this is certainly another possible execution of Algorithm CHC. As we said, graph H represents all possible executions of Algorithm CHC starting from an initial configuration. We do not really need to compute H, it is just intended for analysis purposes. The next theorem exploits the ideal graph H in order to show that Algorithm CHC always terminates and solves the Coating problem.

▶ **Theorem 3.** *Given a convex object obj and $n \geq |EP|$* CONTRACTED *particles forming a connected configuration, Algorithm* CHC *terminates within $\Theta(n^2)$ rounds in a simply connected configuration where EP is totally occupied. Moreover, any configuration generated during the execution of* CHC *is connected in the relaxed sense.*

Proof. Initially, there are n CONTRACTED particles that along with possible holes and *obj* form a connected configuration. Let $H = (V, E)$ be the directed graph representing the executions of Algorithm CHC as defined above. We prove the correctness of Algorithm CHC by showing the three following properties:

P1: Evolution. Each vertex in H, excluding those corresponding to final configurations, where CHCP is accomplished, has at least one outgoing edge;

P2: Connectivity. H is connected, and hence any configuration obtained by an expansion dictated by CHC is connected;

P3: Acyclicity. Graph H is acyclic.

Note that, the finite number of particles along with the preservation of the connectivity imply that there exists a finite number of configurations. Hence, the three properties guarantee that a final configuration is reached, eventually, where CHCP is accomplished. In fact, P1 guarantees that from any configuration, a different one is generated apart from final configurations. Moreover, by P2 we have that any generated configuration is connected. Since P3 guarantees that no configuration can be generated twice, any execution always leads to a final configuration.

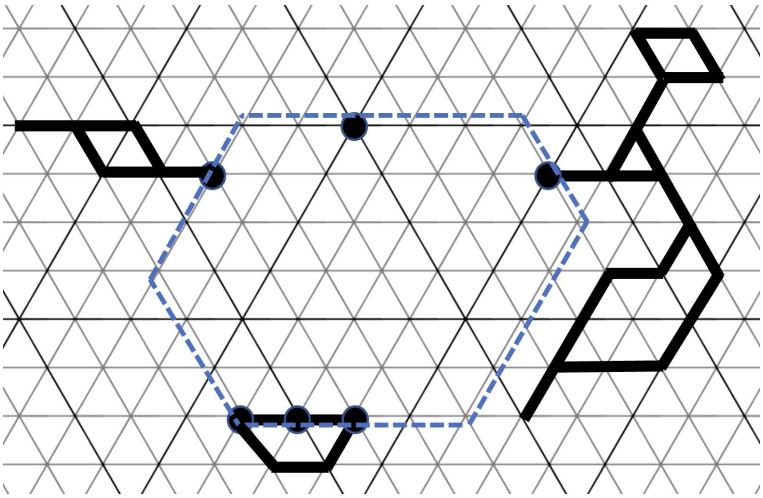


Figure 5 Example of trees of polygons connected via EP, corresponding to the configuration in Figure 3, used in the proof of Theorem 3 for Property P1.

Proof of property P1 (Evolution). If there are semi-occupied nodes, then certainly the configuration will change, eventually. This is due to the fact that EXPANDED particles always complete the committed movement once activated, and due to the fairness of the scheduler. Hence we can assume there are no semi-occupied nodes, i.e., $SO = \emptyset$.

We show that in any (non-final) connected configuration there always exists a CONTRACTED particle p that is allowed to move by algorithm CHC such that:

(i) p is not in EP, $G(N(p, IC))$ is connected and $|N(p, IC)| \leq 3$ or

(ii) p is in EP.

Assume (ii) false and, by contradiction, let us assume each particle p admitting $G(N(p, IC))$ connected, such that $|N(p, IC)| \geq 4$ and hence Algorithm CHC does not allow any expansion. Let us consider the standard planar embedding of the subgraph of lattice G induced by a maximal set of connected CONT and IN nodes, and the envelope containing all nodes of such an embedding. By construction, there might be many of such subgraphs that all together along with *obj* form a connected configuration, see Figure 5.

Since each considered subgraph is connected, the shape of the corresponding envelope is given by a "tree of polygons", i.e., a set of polygons that are connected by paths of straight lines, possibly of length 0 (i.e., connected via one single point corresponding to a single particle). Each tree is connected ("rooted") to *obj* – apart for the special cases provided by configurations like in Figure 4.a where *obj* is contained within a polygon. Moreover, there might be also the case that the mentioned tree of polygons reduces to just one polygon. By hypothesis, the leaves of the tree are not single particles since this would imply that there is at least a particle with only one neighbor. Now, let us consider a leaf of a tree of polygons, and let us assume that it has m vertices (corresponding to m particles). Note that since this polygon is a leaf of the tree, only one of its vertices is connected to the rest of the tree and any other vertex of the polygon corresponds to a particle that has a connected neighborhood.

By hypothesis, we have that any particle has at least 4 neighbors of type CONT or IN. Then, the interior angle of each vertex in the polygon corresponding to these particles measures at least π. Therefore, the sum of the interior angles of the polygon is at least $(m-1)\pi$, which is a contradiction since it is known that such a sum equals $(m-2)\pi$ in any

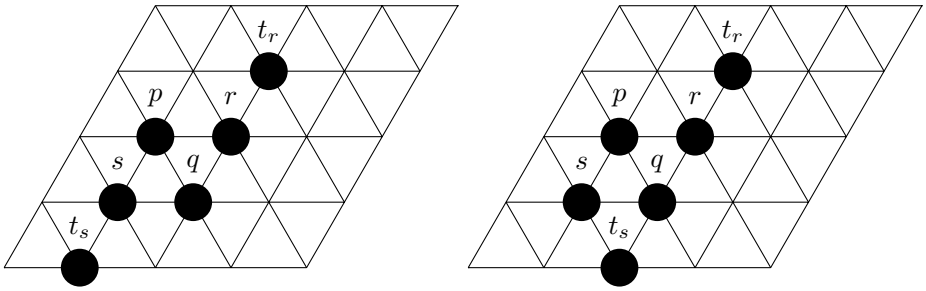


Figure 6 Configurations used in the proof of Theorem 3.

polygon. Hence, we can conclude that there must exist a CONTRACTED particle p such that $G(N(p, IC))$ is connected and $|N(p, IC)| \leq 3$. Furthermore, such a particle belongs to a leaf of a tree of polygons.

Next we show that, given the existence of such a particle, in any non-final configuration where (ii) is false, there exists at least one particle that decides to expand according to Algorithm CHC. In particular, if $|N(p, C)| \leq 2$, then p will expand (see Lines 3-6).

If $|N(p, C)| = 3$, then p is allowed to expand according to three possible conditions dictated by Algorithm CHC. Let r, q and s be the three neighbors of p, with q being the central one. If $|N(r, C)| \leq 2$ ($|N(s, C)| \leq 2$, resp.), then r (s, resp.) will expand, as imposed by Lines 3-6. We remind that we have assumed (ii) false, hence r (s, resp.) is not in EP. By referring to Figures 6.a and 6.b, if $|N(r, C)| > 2$ and $|N(s, C)| > 2$, then r shares neighbor q with p and has a further neighbor, say t_r. Similarly, s is adjacent to p, q, and has a further neighbor t_s. Then two cases can occur: if t_r and t_s are not neighbors of q, then we have a contradiction as the considered polygon is not a leaf of a tree (see Figure 6.a). Otherwise, if at least one among t_r and t_s is a neighbor of q, we have $|N(q, C)| > 3$ and the condition at Line 10 is satisfied (see Figure 6.b), hence p will expand.

About condition at Line 12, it requires that (ii) is true, that is there must exist CONTRACTED particles in EP. If the condition at Line 12 is satisfied, then p expands toward EP (Line 12). Note that this case captures a special occurrence as depicted in Figure 4.c. Basically, the presence of the object requires a special rule in order to not incur in a deadlock. In fact, without such an exception, the particles in the figure would not expand.

If the configuration is non-final but (i) is false and (ii) is true, by the polygon arguments above we have that particles not in EP are all EXPANDED. Moreover, particles cannot be all in EP as otherwise the configuration would be final. We are now going to show that none of the particles at distance 2 from *obj* and adjacent to particles in EP can be expanded by Algorithm CHC toward nodes at distance 3 from *obj*. In so doing, by considering the number of edges among particles at distance 2 from *obj*, we conclude that there must exist at least one particle at distance 2 from *obj* that is EXPANDED toward a particle in EP. Hence, Lines 12-15 are activated and at least one particle in EP expands.

Considering that, for particles not in EP, Algorithm CHC makes particles expanding only toward CONT or IN nodes, let p be a CONTRACTED particle at distance 2 from *obj*, r be a CONTRACTED neighbor of p at distance 3 from *obj* and q be a CONTRACTED neighbor of p in EP. By referring to Figure 7, the first three configurations concern situations where p (the particle at distance 2 from *obj*) cannot expand because $G(N(p, IC))$ is not connected. Whereas, if s is at distance 2 from *obj*, then $G(N(p, IC))$ is connected, hence p would expand toward s. If $|N(p, IC)| > 3$ then p cannot expand. This concludes the proof of Property P1.

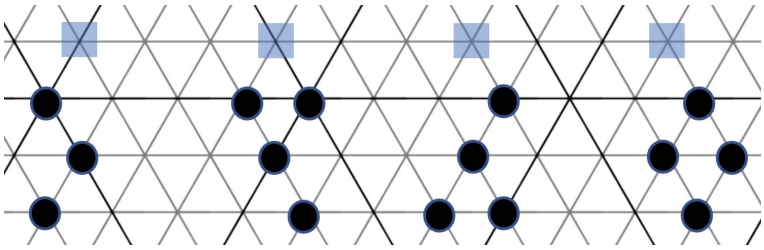


Figure 7 Four possible cases occurring with respect to a particle at distance 2 from the object. In the first three cases, the particle in the middle cannot expand since its neighborhood of CONTRACTED particles is not connected whereas in the last case, Algorithm CHC allows the particles at distance 2 from the object to expand toward each other and, if both activated, by the model only one succeeds.

Proof of property P2 (Connectivity of H). According to Algorithm CHC, we have to analyze when a particle is EXPANDED and whether this (or a combination of such moves) may cause a disconnection.

In fact, the algorithm has been designed to mimic an erosion process from the outer part of the configuration toward obj, plus an oriented movement along EP.

In particular, a particle p not in EP moves toward an IN node v only if p is the central neighbor of the three neighbors of v, say r, p and q, occupied by particles. By the polygon property exploited before, we know that whenever there is a hole, there must exist also a particle in the conditions of p. The movement of p by itself cannot disconnect the configuration. However, this may happen, in principle, because before p actually moves, r and/or q have moved (due to the asynchrony). However, our algorithm guarantees that if r or q move, they can only do it toward the same node v, competing with p.

Movements toward OUT nodes are not allowed, but again a node v may become of type OUT while p has already decided to move toward it. If this is the case, we are guaranteed by the algorithm that the particle that left v won't move again as long as p makes v semi-occupied (see Line 2). Similarly, movements toward nodes occupied by particles cannot cause disconnections.

If $p \in EP$, then p is expanded by the algorithm only if there is another particle expanded toward p, hence the node where p resides may become at most semi-occupied but never purely empty (i.e., without particles expanded toward it), hence again maintaining the configuration connected.

This concludes the proof of Property P2, i.e., H is connected.

Proof of property P3 (Acyclicity of H). Let us associate to each vertex u of H (representing configuration C_u) a pair (α_u, β_u), where α_u is the sum of the number of CONT and IN nodes plus $|SO|$ plus the number of empty nodes in EP in C_u, while β_u is the number of IN nodes plus $|SO|$ plus the number of empty nodes in EP in C_u. We observe that, for each edge (u, v) of H, (α_v, β_v) is lexicographically smaller than (α_u, β_u): in fact, after an action is performed, either at least a CONTRACTED particle in C_u is EXPANDED in C_v (hence the number of CONT nodes decreases) or at least an EXPANDED particle in C_u becomes CONTRACTED in C_v (hence $|SO|$ or the number of IN nodes or the number of empty nodes in EP decreases by at least $k \geq 1$ but the number of CONT nodes increases of the same amount k). Therefore, it is possible to define a topological ordering of the nodes of H as a linear extension of the partial ordering given by the pair (α_i, β_i) of the corresponding configurations C_i.

This concludes the proof of Property P3, i.e., H is acyclic.

Time complexity of Algorithm CHC. About the complexity of the algorithm, each path in H that starts from the node corresponding to the initial configuration and ends in a node corresponding to a final configuration, has a length bounded by the sum of the initial number of CONTRACTED particles plus the number of movements required to occupy EP plus the number of movements necessary to " fill" the initial IN nodes.

The first factor is exactly n. For the second factor, one may think about particles initially disposed in a path shape configuration, touching the object just in one place (say the " head" of the path). Then, the advancement of one step along EP (i.e., toward the final configuration) requires the movement of all the particles in order to not loose connectivity, and consequently the direction toward the object. However, the (adversarial) scheduler may activate all the particles concurrently. In so doing, a whole round is consumed but just one particle expands, namely the " tail" . It follows that when $|EP| = n$ then $O(n^2)$ rounds are required. For the third factor, the number of movements necessary to " fill" the initial IN nodes is upper bounded by $O(n^2)$ with respect to the proposed strategy.

Since an edge of H may correspond to a round of a generic execution, the theorem follows. ◀

Intuitively, the $O(n^2)$ cost of Algorithm CHC in terms of number of rounds turns out to be necessary. In particular, the arguments given for the case of particles disposed in a path shaped configuration provided in the proof of the above theorem turns out to be rather general, i.e., holding for any possible strategy within SILBOT. In fact, when the particles are all aligned and touching the object just in one place, they are forced to proceed one after the other while moving along EP. Any other movement may incur in disconnecting the particles and, or in loosing the direction toward the object, or even worst, in a deadlock with particles expanded in opposite directions. As described above, the sequential movement might be accomplished one step per round. We can then state the following:

▶ **Theorem 4.** *Algorithm* CHC *requires* $\Theta(n^2)$ *rounds for solving the Coating and Holes Compaction problem.*

5 Conclusion

In this paper, we considered the SILBOT model to manage programmable matter. In particular, particles do not have any direct means of communication (i.e., via messages or pebbles), nor local or shared memory, and each particle can retrieve information about its surrounding just up to distance of 2 hops. Moreover, the system is totally asynchronous. This means that a particle can be activated at any time, possibly simultaneously with other particles, and also its actions of expansion and contraction can take different time. All such a freedom is left to the hands of an ideal adversarial scheduler whose only constraint is a kind of fairness, i.e., each particle must be activated within finite time, infinitely often. Finally, particles can only decide on expansions, whereas contractions are automatic and depend on the activations dictated by the scheduler. By adding chirality to the particles we have been able to solve the Coating and Hole Compaction problem. The requirement is to make particles move so as to suitably surround a given static object while obtaining a simply connected configuration. We proposed an optimal algorithm that starting from any initial connected configuration of n particles, eventually leads to solve the problem within $\Theta(n^2)$ rounds.

As future work, one may investigate whether the current assumptions can be further reduced or if other basic tasks can be approached within the same setting. A variant of the Coating problem requires that all the particles contribute to surround the object in a multi-layer fashion. This seems unfeasible within SILBOT with the current setting and gives rise to the investigation on the minimal assumptions for approaching it.

References

1. Hossein Ahmadzadeh, Ellips Masehian, and Masoud Asadpour. Modular robotic systems: Characteristics and applications. *J. Intell. Robotic Syst.*, 81(3-4):317–357, 2016. doi:10.1007/S10846-015-0237-8.
2. Rida A. Bazzi and Joseph L. Briones. Deterministic leader election in self-organizing particle systems. In *Proceedings of the 21st International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 11914. Springer, 2019. doi:10.1007/978-3-030-34992-9_3.
3. Ahmad Reza Cheraghi and Kalman Graffi. A leader based coating algorithm for simple and cave shaped objects with robot swarms. In *5th Asia-Pacific Conference on Intelligent Robot Systems, ACIRS 2020, Singapore, July 17-19, 2020*, pages 43–51. IEEE, 2020. doi:10.1109/ACIRS49895.2020.9162610.
4. Ahmad Reza Cheraghi, Gorden Wunderlich, and Kalman Graffi. General coating of arbitrary objects using robot swarms. In *5th Asia-Pacific Conference on Intelligent Robot Systems, ACIRS 2020, Singapore, July 17-19, 2020*, pages 59–67. IEEE, 2020. doi:10.1109/ACIRS49895.2020.9162617.
5. Anders Lyhne Christensen. *Self-Reconfigurable Robots - An Introduction. Artif. Life*, 18(2):237–240, 2012. doi:10.1162/ARTL_R_00061.
6. Serafino Cicerone, Alessia Di Fonso, Gabriele Di Stefano, and Alfredo Navarra. MOBLOT: molecular oblivious robots. In Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé, editors, *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pages 350–358. ACM, 2021. doi:10.5555/3463952.3463998.
7. Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Gathering of robots on meeting-points: feasibility and optimal resolution algorithms. *Distributed Computing*, 31(1):1–50, 2018. doi:10.1007/S00446-017-0293-3.
8. Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Asynchronous arbitrary pattern formation: the effects of a rigorous approach. *Distributed Computing*, 32(2):91–132, 2019. doi:10.1007/S00446-018-0325-7.
9. Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Embedded pattern formation by asynchronous robots without chirality. *Distributed Computing*, 32(4):291–315, 2019. doi:10.1007/S00446-018-0333-7.
10. Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. "Semi-Asynchronous": a new scheduler in distributed computing. *IEEE Access*, 9, 2021. URL: https://ieeexplore.ieee.org/document/9373406, doi:10.1109/ACCESS.2021.3064880.
11. Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Solving the pattern formation by mobile robots with chirality. *IEEE Access*, 9:88177–88204, 2021. doi:10.1109/ACCESS.2021.3089081.
12. Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. A structured methodology for designing distributed algorithms for mobile entities. *Information Sciences*, 574:111–132, 2021. doi:10.1016/J.INS.2021.05.043.
13. Serafino Cicerone, Alessia Di Fonso, Gabriele Di Stefano, and Alfredo Navarra. Molecular oblivious robots: A new model for robots with assembling capabilities. *IEEE Access*, 11:15701–15724, 2023. doi:10.1109/ACCESS.2023.3244844.

14 Sonia Contera. *Nano Comes to Life: How Nanotechnology Is Transforming Medicine and the Future of Biology*. Princeton University Press, 2019.

15 Gianlorenzo D'Angelo, Mattia D'Emidio, Shantanu Das, Alfredo Navarra, and Giuseppe Prencipe. Asynchronous silent programmable matter achieves leader election and compaction. *IEEE Access*, 8:207619–207634, 2020. doi:10.1109/ACCESS.2020.3038174.

16 Gianlorenzo D'Angelo, Mattia D'Emidio, Shantanu Das, Alfredo Navarra, and Giuseppe Prencipe. Leader election and compaction for asynchronous silent programmable matter. In *Proc. 19th Int.'l Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 276–284. International Foundation for Autonomous Agents and Multiagent Systems, 2020. doi:10.5555/3398761.3398798.

17 Gianlorenzo D'Angelo, Gabriele Di Stefano, and Alfredo Navarra. Gathering on rings under the look-compute-move model. *Distributed Computing*, 27(4):255–285, 2014. doi:10.1007/S00446-014-0212-9.

18 Gianlorenzo D'Angelo, Gabriele Di Stefano, Alfredo Navarra, Nicolas Nisse, and Karol Suchan. Computing on rings by oblivious robots: A unified approach for different tasks. *Algorithmica*, 72(4):1055–1096, 2015. doi:10.1007/S00453-014-9892-6.

19 Gianlorenzo D'Angelo, Alfredo Navarra, and Nicolas Nisse. A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distributed Computing*, 30(1):17–48, 2017. doi:10.1007/S00446-016-0274-Y.

20 Joshua J. Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17(1):81–96, 2018. doi:10.1007/S11047-017-9658-6.

21 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.

22 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: amoebot - a new model for programmable matter. In *Proc. 26th ACM Symp. on Parallelism in Algorithms and Architectures, (SPAA)*, pages 220–222. ACM, 2014. doi:10.1145/2612669.2612712.

23 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal coating for programmable matter. *Theor. Comput. Sci.*, 671:56–68, 2017. doi:10.1016/J.TCS.2016.02.039.

24 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 *Proceedings of 21st International Conf. on DNA Computing and Molecular Programming (DNA)*, volume 9211, pages 117–132. Springer, 2015. doi:10.1007/978-3-319-21999-8_8.

25 Toshio Fukuda and Yoshio Kawauchi. Cellular robotic system (CEBOT) as one of the realization of self-organizing intelligent universal manipulator. In *Proc. of the 1990 IEEE Int.'l Conf. on Robotics and Automation, Cincinnati, Ohio, USA, May 13-18, 1990*, pages 662–667. IEEE, 1990. doi:10.1109/ROBOT.1990.126059.

26 Yonghwan Kim, Yoshiaki Katayama, and Koichi Wada. Pairbot: A novel model for autonomous mobile robot systems consisting of paired robots. *CoRR*, abs/2009.14426, 2020. arXiv:2009.14426.

27 David G. Kirkpatrick, Irina Kostitsyna, Alfredo Navarra, Giuseppe Prencipe, and Nicola Santoro. Separating bounded and unbounded asynchrony for autonomous robots: Point convergence with limited visibility. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 9–19. ACM, 2021. doi:10.1145/3465084.3467910.

28 Ralf Klasing, Adrian Kosowski, and Alfredo Navarra. Taking advantage of symmetries: Gathering of many asynchronous oblivious robots on a ring. *Theor. Comput. Sci.*, 411:3235–3246, 2010. doi:10.1016/J.TCS.2010.05.020.

29 Irina Kostitsyna, David Liedtke, and Christian Scheideler. Universal Coating in the 3D Hybrid Model, 2023. arXiv:2303.16180.

30 Alfredo Navarra and Francesco Piselli. Brief announcement: Line formation in silent programmable matter. In *Proc. 37th Int.'l Symp. on Distributed Computing (DISC)*, volume 281 of *LIPIcs*, pages 45:1–45:8, 2023. doi:10.4230/LIPICS.DISC.2023.45.

31 Alfredo Navarra and Francesco Piselli. Asynchronous silent programmable matter: Line formation. In *Proc. 25th Int.'l Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 14310 of *LNCS*, pages 598–612, 2023. doi:10.1007/978-3-031-44274-2_44.

32 Alfredo Navarra, Giuseppe Prencipe, Samuele Bonini, and Mirco Tracolli. Scattering with programmable matter. In *Proceedings of 37th International Conf. on Advanced Information Networking and Applications (AINA)*, Advances in Intelligent Systems and Computing, pages 236–247. Springer, 2023. doi:10.1007/978-3-031-29056-5_22.

33 Benoît Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Auton. Robots*, 42(8):1619–1633, 2018. doi:10.1007/S10514-018-9710-0.

34 Madhuri Sharon, Angelica SL Rodriguez, Chetna Sharon, and Pio Sifuentes Gallardo. *Nanotechnology in the Defense Industry: Advances, Innovation, and Practical Applications*. John Wiley & Sons, 2019.

35 Pierre Thalamy, Benoît Piranda, and Julien Bourgeois. Engineering efficient and massively parallel 3d self-reconfiguration using sandboxing, scaffolding and coating. *Robotics Auton. Syst.*, 146:103875, 2021. doi:10.1016/J.ROBOT.2021.103875.

36 Tommaso Toffoli and Norman Margolus. Programmable matter: Concepts and realization. *Physica D: Nonlinear Phenomena*, 47(1):263–272, 1991. doi:10.1016/0167-2789(91)90296-L.

37 Thadeu Tucci, Benoît Piranda, and Julien Bourgeois. A distributed self-assembly planning algorithm for modular robots. In Elisabeth André, Sven Koenig, Mehdi Dastani, and Gita Sukthankar, editors, *Proc. of the 17th Int.'l Conf. on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 550–558. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018. URL: http://dl.acm.org/citation.cfm?id=3237465.

38 Mark Yim, Wei min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S. Chirikjian. Modular self-reconfigurable robot systems [Grand Challenges of Robotics]. *IEEE Robotics Automation Magazine*, 14:43–52, 2007. doi:10.1109/MRA.2007.339623.

Tight Bounds on the Message Complexity of Distributed Tree Verification

Shay Kutten
Technion – Israel Institute of Technology, Haifa, Israel

Peter Robinson
Augusta University, GA, USA

Ming Ming Tan
Augusta University, GA, USA

Abstract

We consider the message complexity of verifying whether a given subgraph of the communication network forms a tree with specific properties both in the KT_ρ (nodes know their ρ-hop neighborhood, including node ids) and the KT_0 (nodes do not have this knowledge) models. We develop a rather general framework that helps in establishing tight lower bounds for various tree verification problems. We also consider two different verification requirements: namely that *every* node detects in the case the input is incorrect, as well as the requirement that *at least one* node detects. The results are stronger than previous ones in the sense that we assume that each node knows the number n of nodes in the graph (in some cases) or an α approximation of n (in other cases). For spanning tree verification, we show that the message complexity inherently depends on the quality of the given approximation of n: We show a tight lower bound of $\Omega(n^2)$ for the case $\alpha \geq \sqrt{2}$ and a much better upper bound (i.e., $O(n \log n)$) when nodes are given a tighter approximation. On the other hand, our framework also yields an $\Omega(n^2)$ lower bound on the message complexity of verifying a minimum spanning tree (MST), which reveals a polynomial separation between ST verification and MST verification. This result holds for randomized algorithms with perfect knowledge of the network size, and even when just one node detects illegal inputs, thus improving over the work of Kor, Korman, and Peleg (2013). For verifying a d-approximate BFS tree, we show that the same lower bound holds even if nodes know n exactly, however, the lower bounds is sensitive to d, which is the stretch parameter. First, under the KT_0 assumption, we show a tight message complexity lower bound of $\Omega(n^2)$ in the LOCAL model, when $d \leq \frac{n}{2+\Omega(1)}$. For the KT_ρ assumption, we obtain an upper bound on the message complexity of $O(n \log n)$ in the CONGEST model, when $d \geq \frac{n-1}{\max\{2,\rho+1\}}$, and use a novel charging argument to show that $\Omega\left(\frac{1}{\rho}\left(\frac{n}{\rho}\right)^{1+\frac{c}{\rho}}\right)$ messages are required even in the LOCAL model for comparison-based algorithms. For the well-studied special case of KT_1, we obtain a tight lower bound of $\Omega(n^2)$.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Graph Algorithm, Lower Bound

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.26

Funding *Shay Kutten*: This research was supported in part by grant 2070442 from the ISF.

1 Introduction

Verifying the correctness of a given solution to a graph problem is an important problem with numerous applications. In this setting, there are n nodes that communicate via message passing over the edges of some arbitrary synchronous communication network G. Certain edges in G are labeled and the labeling of an edge e is part of the initial state of the nodes incident to e. For example, when considering the verification of a minimum spanning tree (MST), the label could indicate whether e is part of the MST, whereas for verifying a breadth-first search (BFS) tree, e's label may indicate the direction of the edge in the BFS tree T in addition to whether $e \in T$.

Generally, for a graph verification problem $\mathcal{P}$, we assume that the labels correspond to some (possibly weighted and/or directed) graph structure L of G. After observing the labels of their incident edges, the nodes may exchange messages with their neighbors and, eventually, every node needs to output 1 ("accept") if L corresponds to a *legal solution* to $\mathcal{P}$ for the communication network G. On the other hand, if L is *illegal*, we study two different requirements:

1. All-Detect: Every node outputs 0 ("reject").
2. One-Detects: At least one node outputs 0, whereas the other nodes may output either 0 or 1.

We point out that All-Detect and One-Detects have both been assumed in previous works. The One-Detects requirement, originally proposed in [1], is the basis of the area of distributed verification, e.g., see [19, 8, 25], whereas [17], which is closely related to the results of our work, considers All-Detect. We refer the reader to the survey of [7] for a more thorough survey of these results.

Apart from the spanning tree (ST) and minimum spanning tree (MST) verification problems, we also consider verifying approximate versions of breadth-first search (BFS) trees, which we define next: Consider a connected graph G and some subgraph $H \subseteq G$. We define $\mathsf{dist}_H(u, v)$ to denote the minimum hop distance of nodes u and v, when using only edges in H.

▶ **Definition 1.** *A spanning tree T is a d-*approximate Breadth-first Search Tree (d-approximate BFS) *of G if, for a designated root node u_0, the stretch of the shortest-path distance between u_0 and any other node in T is at most d. Formally, $\mathsf{dist}_T(u_0, v) \leq d \cdot \mathsf{dist}_G(u_0, v)$ for all nodes v in G.*

The input labeling for this problem is similar to that of BFS: it induces a directed subgraph and the labels should indicate, for each node, which one of its ports points to its parent and its children in T (if any).

Closely related to our work are the results of [17], which show that any deterministic distributed algorithm that verifies a minimum spanning tree without any knowledge of the network size and guarantees All-Detect must send $\Omega(m)$ messages in the worst case. We emphasize that the knowledge of n often strengthens algorithms and reduces their complexity, so our results that are given in spite of assuming nodes posses such knowledge (or, sometimes, the knowledge of an approximation of n) are stronger. The work of [30] proves that checking whether a given set of edges induces a spanning tree requires $\Omega(\sqrt{n} + \mathsf{Diam})$ rounds in the CONGEST model even for randomized algorithms and in fact, they prove that the same bound holds for a long list of fundamental graph verification and construction problems. Since [17] also gives a deterministic algorithm that has worst-case complexities of $\tilde{O}(\sqrt{n} + \mathsf{Diam})$ rounds, the time complexity of spanning tree verification is completely resolved, up to logarithmic factors. While the time complexity of distributed verification problems has been studied extensively in previous works, far less is known about the best possible bounds on the message complexity. To the best of our knowledge, the only result on the message complexity of verifying a tree problem was given in the aforementioned work of [17], where they prove that $\Theta(m)$ messages are essentially tight for MST verification. However, their lower bound technique only holds for deterministic algorithms that do not use any knowledge of the network size.

1.1 Our Contributions

We show almost-tight bounds on the message complexity of distributed verification for spanning trees and d-approximate breadth-first search trees. All our lower bounds for the clean network model (i.e. KT_0) hold even in the LOCAL model, whereas the upper bounds work in the CONGEST model.

- In Section 2, we present a general lemma for deriving message complexity lower bounds for graph verification problems under the KT_0 assumption, where nodes are unaware of their neighbors' IDs initially, which may be of independent interest.
- For **spanning tree (ST) verification**, we show that the knowledge of the network size n is crucial for obtaining message-efficient verification algorithms:
  - When nodes know the *exact* network size, we give a deterministic ST verification algorithm that guarantees the strong All-Detect property with a message complexity of only $O(n \log n)$ messages (see Theorem 19 on page 15).
  - If nodes have an α-approximation of n, for any $1 < \alpha < \sqrt{2}$, we still obtain $O(n \log n)$ messages, although we can only achieve One-Detects (i.e., at least one node detects illegal inputs). We prove that this is unavoidable by showing that All-Detect requires $\Omega(n^2)$ messages for any $\alpha > 1$ (see Theorem 10 on page 8).
  - On the other hand, we show that, when $\alpha \geq \sqrt{2}$, there is no hope for obtaining a message-efficient algorithm that guarantees One-Detects, as we prove that there are graphs with $\Theta(n^2)$ edges, where the message complexity is $\Omega(n^2)$.
- For **MST verification**, we show that $\Omega(n^2)$ poses an insurmountable barrier, as it holds for randomized algorithms, when nodes have perfect knowledge of the network size, and even under the weak requirement that just one node detects illegal inputs.
- For **verifying a *d*-approximate BFS tree**, we obtain the following results:
  - Under the KT_0 assumption, we prove that any randomized verification algorithm must send $\Omega(n^2)$ messages (Theorem 6 on page 6), for any $d \leq \frac{n}{2+\Omega(1)}$, even if the nodes have perfect knowledge of the network size. This bound is essentially tight in terms of the stretch d, as we also give an efficient algorithm in Section 5 that achieves $O(n \log n)$ messages, when $d \geq \frac{n}{2} - \frac{1}{2}$.
  - We also consider the d-approximate BFS verification problem under the KT_ρ assumption, for $\rho \geq 1$, where nodes are aware of their ρ-hop neighborhood initially, excluding the private random bits of the nodes. When d is small, we show that the lower bound of $\Omega(n^2)$ continues to hold in KT_1 for comparison-based algorithms. For $\rho \geq 2$ and any $d \leq O\left(\frac{n}{4\rho-2}\right)$, we develop a novel charging argument to show that $\Omega\left(\frac{1}{\rho}\left(\frac{n}{\rho}\right)^{1+\frac{c}{\rho}}\right)$ messages are required, which may turn out to be useful for proving lower bounds for other graph problems in KT_ρ, in particular, for $\rho \geq 2$ (see Theorem 12). We also show that the restriction on d cannot be improved substantially, by giving an upper bound of $O(n \log n)$ messages when $d \geq \frac{n-1}{\max\{2,\rho+1\}}$ (see Theorem 21 on page 15).

1.2 Additional Related Work

The research on ST, MST and BFS is too vast to provide a comprehensive survey. The d-approximate BFS tree problem is an important but limited (to a single source) version of the heavily studied spanner concept [29] of a subgraph with a few edges over which the distance of routing (here, just from the source) is an approximation of the original distance. The study of BFS approximation (in KT_0) has been motivated by the potential saving in the message and time complexities, especially when compared to those of the Bellman-Ford algorithm; see e.g., [2, 24, 6, 21, 12].

Tarjan [31, 14, 11, 5] considered the question of verifying a minimum weight spanning tree (MST) in the context of centralized computing, [16] addressed the problem in the context of PRAM, and [18] studied this question in the context of distributed computing with non-determinism, or with pre-processing. A verification algorithm may be a part of a fault-tolerant algorithm. Specifically, a verification algorithm can be executed repeatedly. If at some point, the verification fails, then an algorithm for re-computation is activated, followed again by repeated activations of the verification algorithm. In the context of self-stabilization, this was suggested in [13, 1] (the algorithms here are not self-stabilizing, though). More generally, in complexity theory and in cryptography, the issue of the complexity of verifying vs. that of computing is one of the main pillars of complexity theory, see, for example, the example of NP-hardness, Zero Knowledge, PCP, and IP (Interactive Proofs). In recent years there has been a lot of research in trying to adapt this kind of theory to distributed computing. We refer the reader to [19, 8, 25] for a more thorough survey of these results. It seems that the idea to verify a program while it is already running (as opposed to methods such as theorem proving, model checking, or even testing) appeared in general computing possibly after they were studied in distributed computing, but meanwhile, this has become a very developed area, see e.g. [22].

1.3 Preliminaries

We consider the standard synchronous CONGEST and LOCAL models [28], where all nodes are awake initially and communicate via message passing. Our main focus of this work is on the *message complexity* of distributed algorithm, which, for deterministic algorithms, is the worst-case number of messages sent in any execution. We assume that each node has a unique ID that is chosen from some polynomial range of integers.

When considering message complexity, the initial knowledge of the nodes becomes important: We follow the standard assumptions in the literature, which are KT_0, in the case where nodes do not know the IDs of their neighbors initially. Under the KT_0 assumption [3], which is also known as the *clean network model* [28], a node u that has δ neighbors also has bidirectional *ports* numbered $1, \ldots, \delta$ over which it can send messages; however, u does not know to which IDs its ports are connected to until it receives a message over this port. In contrast, the KT_1 assumption ensures that each node knows in advance the IDs of its neighbors and the corresponding port assignments. While it takes just a single round to extend KT_0 knowledge to KT_1, this would have required $\Omega(m)$ messages in general. Several algorithms have exploited the additional knowledge provided by KT_1 for designing message-efficient algorithms (e.g., [23, 15]).

2 A Framework for Message Complexity Lower Bounds in the KT_0 LOCAL Model

In this section, we present a general framework for deriving lower bounds on the message complexity of verification problems in the KT_0 LOCAL model.

We remark that the general framework is inspired by the bridge crossing lower bound of [20, 26, 27]. which, however, were designed for specific graph construction and election problems and do not apply to graph *verification*. We give some fairly general requirements for a hard graph and a corresponding labeling (see Definition 4) that, if satisfied, automatically yield nontrivial message complexity lower bounds. We start by introducing some technical machinery.

Rewireable Graphs, Rewirable Components, and Important Edges. Let H be a graph and $V(H)$ denotes that set of nodes in H. We use the notation $H[S]$ to denote the subgraph induced by a subset of nodes $S \subseteq V(H)$, and also define $L[S]$ to denote the labeling restricted to graph $H[S]$. We say that an n-node graph H is *rewirable* if there exist disjoint subsets $A_1, A_2 \subseteq V(H)$ such that $H[A_1]$ and $H[A_2]$ each contains at least an edge, but there are no edges between A_1 and A_2. We call $H[A_1]$ and $H[A_2]$ the *rewirable components* of H.

In our lower bounds, we identify two *important edges* $e_1 = (u_1, v_1) \in H[A_1]$ and $e_2 = (u_2, v_2) \in H[A_2]$. We define H^{e_1,e_2} to be the *rewired graph of* H on the same vertex set, by removing e_1 and e_2, and instead connecting A_1 and A_2 via these four vertices. Concretely, we have $E(H^{e_1,e_2}) = (E(H) \setminus \{e_1, e_2\}) \cup \{(u_1, v_2), (u_2, v_1)\}$, whereby (u_1, v_2) and (u_2, v_1) are connected using the same port numbers in H^{e_1,e_2} as for e_1 and e_2 in H.

▶ **Lemma 2.** *Let H be a rewirable graph. Then each node in H has an identical initial state in both H and H^{e_1,e_2} where H^{e_1,e_2} is any rewired graph of H.*

The proof is straightforward since rewiring does not change the degrees of the nodes and we are considering KT_0, where each node does not have information on the IDs of its neighbors.

We define $\mathsf{Inp}(G, L, \tilde{n})$ to denote the *input* where we execute the algorithm on graph G with the labeling L, and equip all nodes with the network size approximation $\tilde{n}$. An input $\mathsf{Inp}(G, L, \tilde{n})$ is said to be *legal* for problem $\mathcal{P}$ if L is a legal solution to $\mathcal{P}$ on the graph G. For a given algorithm $\mathcal{A}$, we say that inputs $\mathsf{Inp}(H, L, \tilde{n})$ and $\mathsf{Inp}(H', L', \tilde{n})$ are *indistinguishable for a node u* if u has the same probability distribution over its possible state transitions at the start of every round when $\mathcal{A}$ is executed on input $\mathsf{Inp}(H, L, \tilde{n})$ as it does on input $\mathsf{Inp}(H', L', \tilde{n})$. Formally, we write $\mathsf{Inp}(H, L, \tilde{n}) \cong \mathsf{Inp}(H', L', \tilde{n})$ if this indistinguishability is true for every node in the graph, and we use the notation $\mathsf{Inp}(H, L, \tilde{n}) \overset{S}{\cong} \mathsf{Inp}(H', L', \tilde{n})$ when this holds for every node in some set S. The following is immediate from the definition of indistinguishability and Lemma 2:

▶ **Lemma 3.** *Consider a graph H, a labeling L, and a rewired graph H^{e_1,e_2} of H. Let* Found *be the event that some node sends a message over an important edge, i.e., e_1 or e_2. Then, conditioned on event* $\neg$Found*, it holds that $\mathsf{Inp}(H, L, \tilde{n}) \cong \mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n})$.*

▶ **Definition 4** (Hard Base Graph)**.** *Let $\tilde{n}$ be an α-approximation of the network size. We say that a rewirable graph H is a* hard base graph *for an algorithm $\mathcal{A}$ that solves a verification problem $\mathcal{P}$, if there is a labeling L and disjoint vertex sets S_1 and S_2, where $S_1 \cup S_2 = V(H)$ such that, for any important edges e_1 and e_2, the following properties hold:*

(A) $\mathsf{Inp}(H[S_1], L[S_1], \tilde{n}) \overset{S_1}{\cong} \mathsf{Inp}(H, L, \tilde{n}) \overset{S_2}{\cong} \mathsf{Inp}(H[S_2], L[S_2], \tilde{n})$.
(B) $\mathsf{Inp}(H[S_1], L[S_1], \tilde{n})$ *and* $\mathsf{Inp}(H[S_2], L[S_2], \tilde{n})$ *are legal for problem $\mathcal{P}$.*
(C) $\mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n})$ *is illegal for $\mathcal{P}$.*

Remarks. Before stating our lower bound framework based on Definition 4, we provide some clarifying comments: Note that H does not need to be an admissible input to problem $\mathcal{P}$. In particular, H can be a disconnected graph even though problem $\mathcal{P}$ is defined for connected networks. We emphasize that S_1 and S_2 are not necessarily related to the rewirable components A_1 and A_2, introduced above. The difference is that A_1 and A_2 define where the important edges are selected from, whereas S_1 and S_2 partition $V(H)$ in a way such that both $\mathsf{Inp}(H[S_1], L[S_1], \tilde{n})$ and $\mathsf{Inp}(H[S_2], L[S_2], \tilde{n})$ are legal for problem $\mathcal{P}$ (even though $\mathsf{Inp}(H, L, \tilde{n})$ might not actually be an admissible input to problem $\mathcal{P}$).

Note that Property (A) is only relevant when the hard base graph H consists of multiple (i.e., disconnected) components. This will become apparent when proving a lower bound for d-approximate BFS tree verification in Section 2.1, where $S_1 = V(H)$ and $S_2 = \emptyset$, which makes the conditions on $H[S_2]$ stated in Property (A) and Property (B) vacuously true.

We prove the following result in Appendix A:

▶ **Lemma 5** (General KT_0 Lower Bound)**.** *Consider any ϵ-error randomized algorithm $\mathcal{A}$ for a graph verification problem $\mathcal{P}$, where $\epsilon < \frac{1}{6}$. If there exists a hard base graph H for problem $\mathcal{P}$ such that the rewirable components $H[A_1]$ and $H[A_2]$ are both cliques of size $\Theta(n)$, then $\mathcal{A}$ has an expected message complexity of $\Omega(n^2)$ in the KT_0 LOCAL model.*

2.1 A Lower Bound for d-Approximate BFS Verification

In this section, we assume that the labeling T is an arbitrary directed subgraph of the network G. The verification task requires checking whether T is indeed a d-approximate BFS tree of G (see Def. 1). Since the subgraph T is directed, each node in T knows its parents and children in T. We remark that this explicit specification of the direction of the tree (compared to the case where T is an undirected subtree of G) can only help the algorithm and hence strengthens our lower bound.

▶ **Theorem 6.** *Suppose that $d = \frac{n}{2} - \gamma$, for some $\gamma \leq \frac{n}{2} - 1$. Consider any ϵ-error randomized algorithm that solves d-approximate BFS tree verification in the KT_0 LOCAL model, and for any $\epsilon < \frac{1}{6}$. There exists an n-node network where the message complexity is $\Omega(\gamma^2)$ in expectation under the KT_0 assumption. In particular, for any $d \leq \frac{n}{2+\Omega(1)}$, this yields $\Omega(n^2)$ messages. This holds even when all nodes know the exact network size n.*

In the remainder of this section, we prove Theorem 6. To instantiate Lemma 5, we define a hard base graph H that satisfies Def. 4. For the sake of readability and since it does not change the asymptotic bounds, we assume that $2d$ and γ are integers. We group the vertices of H into $2d+2$ *levels* numbered $0, \ldots, 2d+1$. Levels 1 and $2d+1$ contain γ nodes each, denoted by $u_1^{(1)}, \ldots, u_\gamma^{(1)}$ and $u_1^{(2d+1)}, \ldots, u_\gamma^{(2d+1)}$, respectively. All the other levels consist of only a single node, and we use $u^{(i)}$ to denote the (single) node on level $i \in ([0, 2d] \setminus \{1\})$.

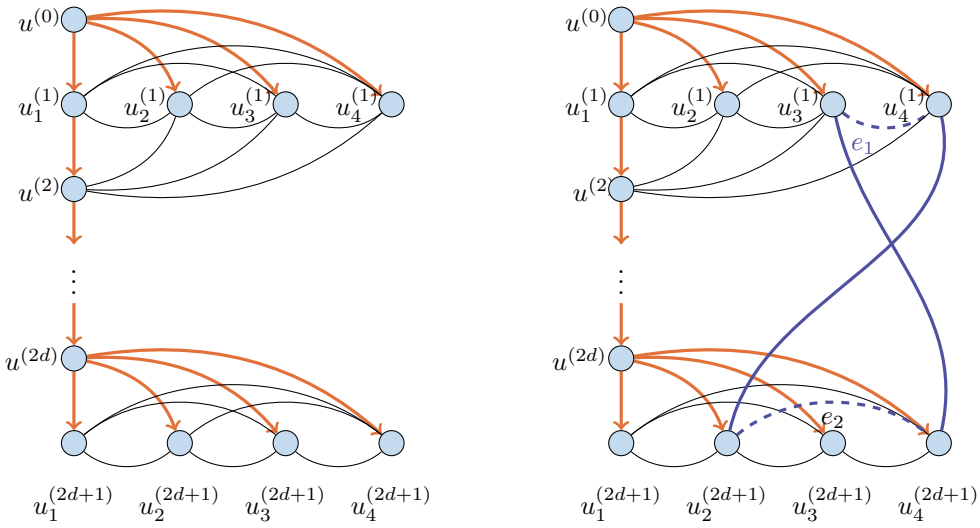
Next, we define the edges of H. Every node on level $i < 2d+2$ is connected via *inter-level* edges to every other node on level $i+1$. Moreover, the nodes on each level form a clique. Figure 1a gives an example of this construction.

To ensure that H is rewirable, we set A_1 to be the clique nodes on level 1 and A_2 to contain the clique nodes on level $2d+1$. We summarize the properties of H in the next lemma:

▶ **Lemma 7.** *Graph H has $2\gamma + 2d = n$ nodes and $\gamma^2 + 2\gamma + 2d - 2 = \Theta(\gamma^2)$ edges. It is a rewirable graph that contains two cliques, each on γ nodes, as rewirable components. The directed subgraph $T \subseteq H$ which contains all nodes of H and all directed inter-level edges is a d-approximate BFS tree of H.*

The next lemma tells us that the properties of Definition 4 hold, which will allow us to instantiate Lemma 5 for obtaining the sought lower bound.

▶ **Lemma 8.** *Consider the directed tree T defined in Lemma 7. Then, T is a d-approximate BFS tree of H, but not of any rewired graph H^{e_1,e_2} and, consequently, graph H is a hard base graph for d-approximate BFS verification with labeling T. This holds even if all nodes know the exact network size.*



(a) The hard base graph H for d-approximate BFS tree verification.

(b) The rewired graph H^{e_1,e_2} with the (dashed) important edges e_1 and e_2 that are replaced by the thick blue edges.

Figure 1 The graphs used in the lower bound for d-approximate BFS verification. The directed orange edges show the labeling, which is a BFS tree, and hence a legal input for any d.

Proof. We define $S_1 = V(H)$ and thus $S_2 = \emptyset$. For a given pair of important edges e_1 and e_2, we need to show Properties (A), (B), and (C) of Definition 4. Property (A) is trivial. For (B), note that $\mathsf{Inp}(H[S_1], L[S_1], n) = \mathsf{Inp}(H, T, n)$, and by Lemma 7, T is indeed a d-approximate BFS tree of H. For property (C), observe that the input tree T is not a d-approximate BFS tree of the rewired graph H^{e_1,e_2}, for any e_1 and e_2, because $\mathsf{dist}_T(u^{(0)}, u_k^{(2d+1)}) = 2d+1$, whereas $\mathsf{dist}_{H^{e_1,e_2}}(u^{(0)}, u_k^{(2d+1)}) = 2$ for $k \geq 1$. ◀

Theorem 6 follows by combining Lemma 7 and Lemma 8 with the general lower bound Lemma 5.

2.2 A Lower Bound for MST Verification

We now show that the lower bound graph used for d-approximate BFS tree verification in Section 2.1 is versatile enough to also yield a lower bound for MST verification.

▶ **Corollary 9.** *Suppose that all nodes know the exact network size. Let $\epsilon > 0$ be a sufficiently small constant. Any ϵ-error randomized algorithm that verifies whether an input is a minimum spanning tree in the* KT_0 LOCAL *model sends $\Omega(n^2)$ messages in expectation, even if the algorithm only ensures that at least one node detects illegal inputs (i.e.* One-Detects*). The same result holds for verifying an approximate MST.*

Proof. We use a variant of the hard base graph H that we employed in the proof of Theorem 6, which is shown in Figure 1. The main difference is that now we consider an input labeling L that induces an undirected weighted subgraph, and we consider only 5 layers, where layer 1 and layer 4 are cliques, and the other layers consist of a single node each. Hence, there is a single edge $e = (u^{(2)}, u^{(3)})$ that acts as a bridge between the two cliques. The edge e has

weight $W > 1$ whereas every other edge in the graph has weight 0. The labeling L that we consider induces an MST that contains e and some arbitrary spanning tree on the rest of the graph. We now show that this satisfies Properties (A), (B), and (C) of Definition 4: We choose $S_1 = V(H)$ and $S_2 = \emptyset$, which makes Property (A) vacuously true. For Property (B), observe that $\mathsf{Inp}(H[S_1], L[S_1], n)$ corresponds to a valid MST of H and hence is legal. To see why Property (C) holds, it is sufficient to observe that any MST of the rewired graph H^{e_1,e_2} must include one of the rewired edges (connecting the two cliques) instead of the bridge edge e that has weight W. ◀

2.3 A Lower Bound for Spanning Tree Verification

We now consider the *Spanning Tree (ST) verification* problem where nodes have an α-approximation of the network size. The input is a connected graph G, a subgraph T of G, and an integer $\tilde{n}$ which is an α-approximation to the actual network size. The verification task requires checking distributively whether T is an ST of G, i.e., T is a tree that contains all nodes.

We remark that a message complexity lower bound of $\Omega(n^2)$ assuming All-Detect was previously shown by Kor, Korman, and Peleg [17] for deterministic algorithms in the setting where the network size is unknown. By using our framework, we generalize the message complexity lower bound to randomized algorithms and to the setting where nodes have an α-approximation of the network size. We also show that the bound depends on whether we assume One-Detects or All-Detect.

▶ **Theorem 10.** *Consider an ϵ-error randomized algorithm $\mathcal{A}$ that solves spanning tree verification in the KT_0 LOCAL model, where $\epsilon < \frac{1}{6}$, and suppose that nodes know an α-approximation of the network size:*

- *If all nodes detect an illegal input (i.e. All-Detect), then the message complexity is $\Omega(n^2)$ in expectation, for any $\alpha > 1$.*
- *If the algorithm satisfies only One-Detects, then the expected message complexity is still $\Omega(n^2)$, for any $\alpha \geq \sqrt{2}$.*

We emphasize that the bounds on the approximation ratio in Theorem 10 are tight, since in Section 4 we show that, if $\alpha < \sqrt{2}$, then the message complexity reduces drastically to just $O(n \log n)$ for One-Detects, and this holds even for All-Detect in the case where $\alpha = 1$.

Combining the following lemma with Lemma 5 immediately yields Theorem 10:

▶ **Lemma 11.** *There exists a graph H that is a hard base graph for ST verification with a suitable labeling where the rewireable components are cliques of size $\Omega(n)$, assuming the following restriction on the network size approximation known by the nodes:*

(i) *For* All-Detect*, this holds for any $\alpha > 1$.*
(ii) *For* One-Detects*, we require that $\alpha \geq \sqrt{2}$.*

Proof. We define a suitable lower bound graph H that consists of two (disconnected) cliques C and C'. We will specify the sizes of C and C' below. To obtain a labeling, we define T and T' be a spanning tree of the subgraphs C and C', respectively. To make H rewirable, we fix $A_1 = V(C)$ and $A_2 = V(C')$, i.e., the important edges will connect the cut (C, C') when being rewired.

We fix the sets $S_1 = V(C)$ and $S_2 = V(C')$ as required by Def. 4. For both cases, (i) and (ii), it is easy to see that Property (A) of Def. 4 holds since C and C' are (disconnected) components.

Thus, we only need to focus on Properties (B) and (C). First, we show Case (i) which is for All-Detect. We define C to be of size t and C' to be of size $(\alpha-1)t$,[1] which means that $n = \alpha t$. We equip nodes with the approximate network size $\tilde{n} = t$. It follows that both $H[A_1]$ and $H[A_2]$ contain $\Theta(n^2)$ edges. Consider the input $\mathsf{Inp}(H[S_1], L[S_1], \tilde{n}) = \mathsf{Inp}(C, T, t)$, which is legal since T is a spanning tree for C. This shows (B), since we consider All-Detect. For Property (C), consider any important edges e_1 and e_2. Graph H^{e_1,e_2} is connected, but the subgraph induced by the labeling $L := T \cup T'$ is not. Recalling that $V(H^{e_1,e_2})$ contains αt nodes, it follows that the network size approximation $\tilde{n} = t$ is indeed an α-approximation and thus the input $\mathsf{Inp}(H^{e_1,e_2}, T \cup T', t)$ is admissible for the algorithm and represents an illegal labeling. This completes the proof of (i).

Next, we show Case (ii) which is for One-Detects. Both C and C' contain exactly t nodes, and hence, again, it follows that H is rewirable. Here, we equip nodes with a network size approximation $\tilde{n} = \alpha t$. To show that Property (B) holds, we only need to prove that the network size approximation is admissible, since the rest of the argument is the same as for (i). Clearly $\mathsf{Inp}(H[S_1], L[S_1], \tilde{n}) = \mathsf{Inp}(C, T, \alpha\, t)$ and $\mathsf{Inp}(H[S_2], L[S_2], \tilde{n}) = \mathsf{Inp}(C', T', \alpha\, t)$ are both admissible and legal since C and C' contain t nodes each. Finally, For Property (C), fix any two important edges e_1 and e_2. We only need to argue that the input $\mathsf{Inp}(H^{e_1,e_2}, T \cup T', \alpha\, t)$ has an admissible network size approximation, as the rest of the argument that shows that it is not a valid spanning tree is the same as for (i). Observe that $n = |V(H^{e_1,e_2})| = 2t$ whereas $\tilde{n} = \alpha \cdot t$. It holds that $\tilde{n} \in [n/\alpha, \alpha\, n]$ if and only if

$$\frac{2t}{\alpha} \leq \tilde{n} = \alpha \cdot t \leq 2\alpha\, t.$$

In particular, the left inequality is true for any $\alpha \geq \sqrt{2}$, as required. ◀

3 A Lower Bound for d-Approximate BFS Verification in KT_ρ ($\rho \geq 1$)

Throughout this section, we consider comparison-based algorithms in the KT_ρ, CONGEST model. Apart from the result for broadcast of [3], we are not aware of any other superlinear (in n) message complexity lower bounds in the KT_ρ setting that hold for $\rho \geq 2$.[2] Even though it is relatively straightforward to generalize our approach to other verification problems such as spanning tree verification, here we exclusively focus on verifying a d-approximate BFS tree, which is more challenging in terms of techniques, as it requires us to exclusively deal with *connected* graphs. In particular, the charging argument in the lower bound approach of [3] crucially exploits the assumption that their "base" graph is disconnected.

In KT_ρ, a node x knows the IDs of all nodes that are at a distance less than or equal to ρ from x, and also knows the adjacencies of all nodes that have a distance of up to $\rho - 1$ from x. When considering the special case KT_1, a node simply knows the IDs of all its neighbors. Formally, we define $S_\rho(x) = \cup_{1 \leq r \leq \rho} C_r(x)$, where $C_r(x)$ consists of all nodes that are at distance r from x. Let $E_\rho(x) = \{(y,z) \in E | y, z \in S_\rho(x)\} \setminus \{(y,z) \in E | y, z \in C_\rho(x)\}$. The *$\rho$-neighborhood* of x is the subgraph $N_\rho(x) = (S_\rho(x), E_\rho(x))$.

We focus on comparison-based algorithms in the KT_ρ CONGEST model, which operate under the following restrictions: We assume that each node u has two types of variables. *ID variables* which contain u's neighborhood information, and *ordinary variables* that are

[1] For the sake of readability, we assume that $(\alpha-1)t$ is an integer. In the case that $(\alpha-1)t$ is not an integer, we set the size of G' to be $\lfloor(\alpha-1)t\rfloor$.

[2] We point out that [3] only provide a full proof of their result for the special case $\rho = 1$.

initialized to 0. During its local computation, a node may compare ID variables and it may perform some arbitrary computation on its ordinary variables. It can store the result of these computations in other ordinary variables. Each message sent by u may include $B = O(1)$ node IDs and it may also contain u's entire list of ordinary variables. Note that these are standard assumptions for comparison-based algorithms (see e.g., [3, 9, 27]). In the remainder of this section, we prove Theorem 12.

▶ **Theorem 12.** *Let $\beta > 0$ and $\epsilon > 0$ be suitable constants. Consider any $\rho \geq 1$, and $d \leq \frac{(1-\beta)n}{4\rho-2}$. For any ϵ-error randomized comparison-based algorithm that solves d-approximate BFS tree verification under the KT_ρ assumption with the guarantee that at least one node detects illegal inputs (i.e., $\mathsf{One\text{-}Detects}$), there exists a network where its message complexity is $\Omega\left(\frac{1}{\rho}\left(\frac{n}{\rho}\right)^{1+\frac{c}{\rho}}\right)$, for some constant $c > 0$. This holds even when all nodes know the exact network size. For the special case $\rho = 1$, we obtain a lower bound of $\Omega(n^2)$.*

The Lower Bound Graph Family

For a given $\rho \geq 1$ and $d \geq 1$, we construct an infinite family of n-node graphs (and their respective rewired variants) that yield the claimed bound of Theorem 12. We consider a graph H of n vertices that resembles the lower bound graph that sufficed for the KT_0 assumption (see Lemma 7). In the KT_ρ setting, a crucial difference is that, in order to show that the initial state of a node x has an order-equivalent ρ-neighborhood in H and in $H^{e,\tilde{e}}$, we need to add $\rho - 1$ layers before and after each of the rewirable components A_1 and A_2.

Figure 2 on page 12 gives an example of the graph construction that we now describe in detail: Let $\ell = (\rho + 1)d + \rho$. The vertices of H are partitioned into $\ell + 1$ *levels* numbered $0, \ldots, \ell$. Let $L = [1, 2\rho - 1]$ and $L' = [\ell - 2\rho + 2, \ell]$. The vertices of levels in L correspond to the nodes of the first $2\rho - 1$ layers after the layer 0. The vertices of levels in L' correspond to the nodes of the last $2\rho - 1$ layers. Let

$$k = \frac{1}{4\rho - 2} - \frac{1}{n} \cdot \frac{d(\rho + 1) - 3\rho + 3}{4\rho - 2} = \frac{1}{4\rho - 2} - O\left(\frac{d}{n}\right), \tag{1}$$

and let $\gamma = kn$.[3] For each $i \in L \cup L'$, level i contains γ nodes each, denoted by $u_1^{(i)}, \ldots, u_\gamma^{(i)}$. All the other levels consist of only a single node, and we use $u^{(i)}$ to denote the (single) node on level i for all $i \in [0, \ell] \setminus \{L \cup L'\}$.

Next, we define the edges of H: There is an edge from the root to every node in level 1. For each $i \in L \setminus \{1\} \cup L' \setminus \{\ell - 2\rho + 2\}$, there is an edge between nodes $u_j^{(i-1)}$ and $u_j^{(i)}$, for $1 \leq j \leq \gamma$. Let $M = [2\rho, \ell - 2\rho + 2]$. For each $i \in M$, all nodes at level i have an edge to every node at level $i - 1$; note that a level may only contain one such node. The nodes at level ρ form a subgraph C_ρ (defined below) and, similarly, the nodes at level $d(\rho + 1) + 1$ form another subgraph C'_ρ which is isomorphic to C_ρ. We call the edges that connect nodes at different levels the *inter-level edges*.

For a node $x \in C_\rho$ (resp. $x \in C'_\rho$), we use the notation $\tilde{x}$ to refer to its copy in C'_ρ (resp. in C_ρ), and we extend this notation to the edges in $C_\rho \cup C'_\rho$ in a natural way. An edge $e = (u, v) \in C_\rho$ induces the *rewired graph* $H^{e,\tilde{e}}$, where $\tilde{e} = (\tilde{u}, \tilde{v})$ is e's copy in C'_ρ. It is

[3] For the sake of readability, we assume that ℓ and γ are integers, which does not affect the asymptotic bounds.

straightforward to verify that each node in H has the same initial state in H and in $H^{e,\tilde{e}}$, since the ρ-neighborhood of each node x in C_ρ is isomorphic to the ρ-neighborhood $\tilde{x}$, by construction.

Let T be the directed tree, which contains all nodes of H and all directed inter-level edges. Then, T is indeed a (1-approximate) BFS tree of H. However, T is not a d-approximate BFS tree of $H^{e,\tilde{e}}$ because $\mathsf{dist}_T(u^{(0)}, u_k^{(d(\rho+1)+1)}) = d(\rho+1)+1$, whereas $\mathsf{dist}_{H^{e,\tilde{e}}}(u^{(0)}, u_k^{(d(\rho+1)+1)}) = \rho+1$ for $k \geq 1$. Hence the algorithm must produce different outputs on these two graphs when the input labeling is T.

The subgraph C_ρ

Intuitively speaking, the achieved message complexity lower bound will be determined by the number of edges in C_ρ, and thus we aim to make C_ρ as dense as possible. For $\rho = 1$, we simply define C_1 (and hence also C_1') to be the clique on γ vertices.

To simplify our argument needed for the lower bound result, we require that the subgraph C_ρ is such that if two nodes have distance at most ρ from each other, then there exists a unique path of that length between the two nodes, which we call the *ρ-unique shortest path* property. This trivially holds for $\rho = 1$ since C_1 and C_1' are cliques. For $\rho \geq 2$, we define C_ρ (and hence also C_ρ') to be a subgraph of γ nodes with girth greater than 2ρ and with $\Omega(\gamma^{1+\frac{c}{\rho}})$ edges for a constant $c > 0$. Recall that the *girth* is the length of the shortest cycle in a graph and that a graph with a girth greater than 2ρ has the ρ-unique shortest path property that we specified earlier. The existence of such graphs C_ρ is known due to [4].

We summarize the properties of H in the following lemma:

▶ **Lemma 13.** *The graph H contains subgraphs C_ρ and C_ρ' which satisfy the ρ-unique shortest path property, and $|V(C_\rho)| = |V(C_\rho')| = \gamma = kn$ such that k satisfies (1). If $\rho = 1$, the subgraph $C_1 \cup C_1'$ has $\Omega(n^2)$ edges. If $\rho \geq 2$, the subgraph $C_\rho \cup C_\rho'$ has $\Omega(\gamma^{1+\frac{c}{\rho}})$ edges for some constant $c > 0$. The directed tree T, which contains all nodes of H and all directed inter-level edges, is a d-approximate BFS tree of H but not of $H^{e,\tilde{e}}$.*

In our analysis, we make use of a particular notion of "connectedness" similar to the notion in [3]:

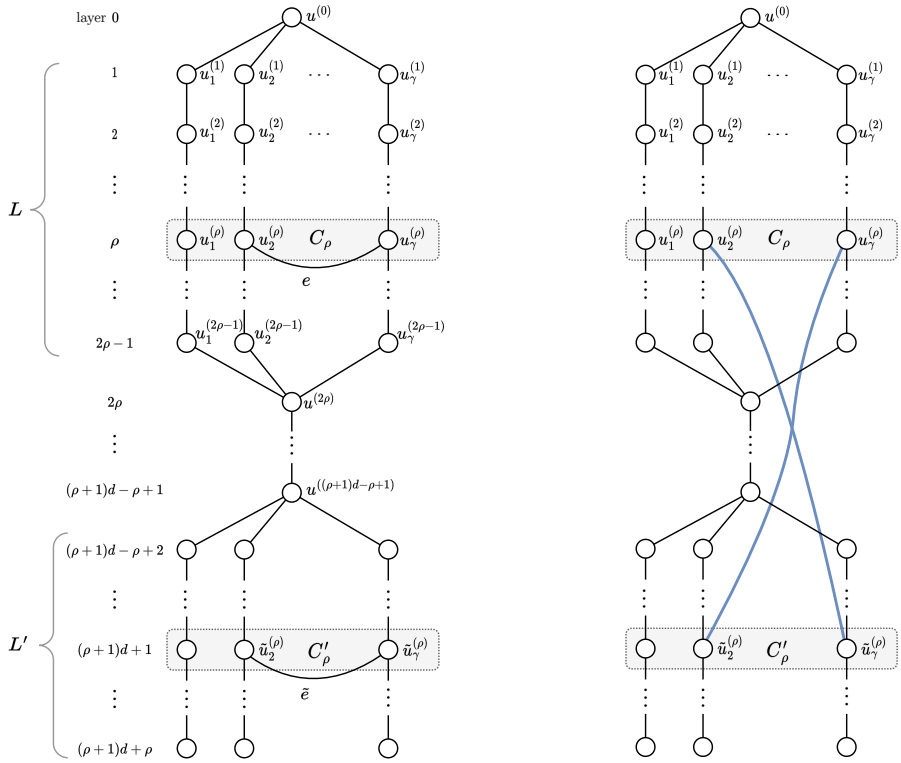
▶ **Definition 14** (e-connected)**.** *Consider an edge e in a graph that satisfies the ρ-unique shortest path property. We say that nodes x and z are* e-connected*, denoted by $x \overset{e}{-} z$, if $\mathsf{dist}(x, z) \leq \rho$ and the (unique) shortest path from x to z contains e.*

As outlined above, the proof of [3], for showing a lower bound on the message complexity of broadcast in KT_ρ, crucially relies on the fact that their base graph is disconnected. For showing a lower bound for the d-approximate BFS tree verification problem, we need the base graph as well as the rewired graphs to be connected. Thus, we need to introduce different rules for *utilizing* and *charging* edges, which have the added benefit of leading to a significantly simplified indistinguishability proof, as we will see in the proof of Lemma 18 below.

▶ **Definition 15** (utilized/unutilized edge)**.** *We say that an edge $e = (u, v) \in C_\rho \cup C_\rho'$ is* utilized *if one of the following conditions hold:*

1. *a message is sent across $e = (u, v)$;*
2. *there are two nodes x and z such that $x \overset{e}{-} z$, and x receives or sends the ID of z or $\tilde{z}$.*

Otherwise, we say that the edge e is unutilized*.*



(a) The base graph H.

(b) The rewired graph $H^{e,\tilde{e}}$.

Figure 2 The graph construction for proving a lower bound in KT_ρ for verifying a d-approximate BFS tree, where we have omitted the input labeling and node IDs. The shaded subgraphs are high girth graphs on γ nodes that have $\Omega\left(\gamma^{1+\frac{c}{\rho}}\right)$ edges. We emphasize that even the base graph H is connected, and thus the approach of [3] does not apply in our setting.

▶ **Definition 16** (charging rule). *If an edge (x, y) is used to send a message that contains the ID of node z, we charge one unit to each of the following edges:*

1. *(x, y);*
2. *for each edge e, such that either $x \overset{e}{-} z$ or $x \overset{e}{-} \tilde{z}$.*
3. *for each edge e, such that either $y \overset{e}{-} z$ or $y \overset{e}{-} \tilde{z}$.*

▶ **Lemma 17.** *Let μ be the number of utilized edges in an execution on H. Then the message complexity of the execution is $\Omega(\mu/\rho)$.*

Proof. Let μ be the number of utilized edges, M be the number of messages sent and C be the total cost charged in the execution. A message sent on an edge (x, y) will incur one unit of charge to (x, y) and for each z where its ID is in the message, a unit of charge is applied to each edge e where $x \overset{e}{-} z$, $x \overset{e}{-} \tilde{z}$, $y \overset{e}{-} z$ and $y \overset{e}{-} \tilde{z}$. Note that there are at most ρ edges of each of these types since the graph H satisfies the ρ-unique shortest path property. Note that each message can carry at most B IDs of other nodes. Consequently, for each of the messages sent, at most $1 + 4B\rho$ edges are charged. Hence, we have $C \leq (1 + 4B\rho)M$. On the other hand, each utilized edge is charged at least once. Hence, $\mu \leq C$. As a result, we have $M \geq \frac{\mu}{1+4B\rho}$ and the lower bound $\Omega(\mu/\rho)$ follows. ◀

Next, we present a suitable ID assignment ϕ for $V(H)$ $(= V(H^{e,\tilde{e}}))$ defined as follows:

1. assign a distinct even integer in $[0, 2(2\rho-1)\gamma]$ to each node in levels $1, \ldots, 2\rho-1$;
2. assign a distinct odd integer in $[1, 2(2\rho-1)\gamma+1]$ to each node x in levels $\ell-2\rho+2, \ldots, \ell$ such that $\phi(\tilde{x}) = \phi(x)+1$;
3. assign arbitrary unique integers from $[2(2\rho-1)\gamma+2, n]$ to the remaining nodes in $V(H)$.

We are now ready to present the indistinguishably result. For now, we focus on deterministic algorithms; we later extend the result to randomized algorithms via a simple application of Yao's lemma.

▶ **Lemma 18.** *Consider a deterministic comparison-based algorithm $\mathcal{A}$, the ID assignment ϕ, graph H and any rewired graph $H^{e,\tilde{e}}$, where $e \in C_\rho$. If e and $\tilde{e}$ are both unutilized (see Def. 15), then H and $H^{e,\tilde{e}}$ are indistinguishable for every node u when executing $\mathcal{A}$, i.e., u has an order equivalent initial state in both networks, and it sends and receives the same sequence of messages in H as it does in $H^{e,\tilde{e}}$.*

Proof. Let $e = (u, v)$. We first show that the statement holds for round 1, which is immediate for any node not in the $(\rho-1)$-neighborhood of either u, v, $\tilde{u}$, or $\tilde{v}$. Thus we focus on these nodes. We denote the set of these nodes as $N = N_{\rho-1}(u) \cup N_{\rho-1}(v) \cup N_{\rho-1}(\tilde{u}) \cup N_{\rho-1}(\tilde{v})$.

First, we show that the initial state of a node $x \in N$ has an order-equivalent $(\rho-1)$-neighborhood in H and in $H^{e,\tilde{e}}$. We observe that the $(\rho-1)$-neighborhood of x is isomorphic (ignoring the ID assignments) to the $(\rho-1)$-neighborhood of $\tilde{x}$. However, the IDs in the $(\rho-1)$-neighborhood of x in H are not the same as the IDs in the $(\rho-1)$-neighborhood of x in $H^{e,\tilde{e}}$. For instance, node u is adjacent to v in H but is adjacent to $\tilde{v}$ in $H^{e,\tilde{e}}$. Nevertheless, it follows from the definition of ϕ that the IDs in the $(\rho-1)$-neighborhood of x in H are order-equivalent to the IDs in the $(\rho-1)$-neighborhood of x in $H^{e,\tilde{e}}$, and thus x has order-equivalent neighborhoods in H and $H^{e,\tilde{e}}$.

Second, we show that each node sends the same messages in round 1. Since x has order-equivalent neighborhoods in both executions and recalling that the algorithm is comparison-based, any ordinary variable that x computes at the start of round 1 must have the same value in both executions. Next, we show that the ID type variable used for x in sending messages is identical in both executions as well. Recall that the IDs that x knows at the start of round 1 are exactly the IDs in its ρ-neighborhood, which may be different in H compared to $H^{e,\tilde{e}}$, as we have observed above. In particular, the IDs in $N_\rho(x)$ of H that are not in $N_\rho(x)$ of $H^{e,\tilde{e}}$ are the IDs of all vertices z, for which it holds that x and z are e-connected. The reason for this is that, in $H^{e,\tilde{e}}$, nodes x and $\tilde{z}$ are e-connected, whereas x and z are not. However, since e and $\tilde{e}$ are unutilized, x can neither include the ID of z in the execution on H, nor the ID of its neighbor $\tilde{z}$ in the execution on $H^{e,\tilde{e}}$, for any node z for which $x \overset{e}{-} z$. Hence, all IDs in the message sent by node x are identical in both executions. Therefore, x sends the same messages in both networks, H and $H^{e,\tilde{e}}$.

Finally, we show that if every node sends the same messages during round $r-1$, then every node sends the same messages in round r. We use induction over the rounds, where the basis already follows from above. Now consider some round $r > 1$ and assume that every node sent the same messages in round $r-1$ in both H and $H^{e,\tilde{e}}$. In this case, every node also receives the same messages during round $r-1$ in both executions. Consider the set of messages M received by x. Since e and $\tilde{e}$ are unutilized, no message in M contains the ID of z or $\tilde{z}$ for all nodes z that x is e-connected to. Hence, all IDs received by x correspond to the same nodes in H and $H^{e,\tilde{e}}$. Again, using the fact that the algorithm is comparison-based and the fact that all IDs that may be contained in the messages in M belong to nodes that have an identical ρ-neighborhood in H and $H^{e,\tilde{e}}$, it follows that x sends the same messages in round r. ◀

3.1 Proof of Theorem 12

First, consider any deterministic comparison-based algorithm. Assume towards a contradiction that there exists a deterministic comparison-based algorithm $\mathcal{A}$ that solves d-approximate BFS verification on input $\mathsf{Inp}(H, T, n)$ with message complexity $o\left(\frac{1}{\rho}\gamma^{1+\frac{c}{\rho}}\right)$. Then by Lemma 17, the number of utilized edges is $o(\gamma^{1+\frac{c}{\rho}})$. Lemma 13 tells us that there are $\Omega(\gamma^{1+\frac{c}{\rho}})$ edges in $C_\rho \cup C'_\rho$. Hence, there exists an edge e from C_ρ such that e and $\tilde{e}$ are unutilized. Consider the rewired graph $H^{e,\tilde{e}}$. Lemma 18 ensures that every node outputs the same result in both executions. However, according to Lemma 13, T is a d-approximate BFS of H but not of $H^{e,\tilde{e}}$, which provides a contradiction.

To obtain the claimed bound, we need to show that $\Omega\left(\frac{1}{\rho}\gamma^{1+\frac{c}{\rho}}\right) = \Omega\left(\frac{1}{\rho}\left(\frac{n}{\rho}\right)^{1+\frac{c}{\rho}}\right)$. By definition,

$$\begin{aligned}
\gamma &= kn \\
\text{(from (1))} \quad &= \frac{n}{4\rho-2} - \frac{d(\rho+1)}{4\rho-2} + \frac{3(\rho-1)}{2(\rho-2)} \\
\text{(since } \frac{\rho-1}{\rho-2} > 1\text{)} \quad &\geq \frac{n}{4\rho-2} - \frac{d(\rho+1)}{4\rho-2} \\
\text{(since } d \leq \frac{(1-\beta)n}{\rho+1}\text{)} \quad &\geq \frac{\beta n}{4\rho-2} = \Omega\left(\frac{n}{\rho}\right).
\end{aligned}$$

Randomized Algorithms. So far, we have restricted out attention to deterministic algorithms that succeed on every input. To extend this result to randomized Monte Carlo algorithms that fail with some small probability ϵ, we follow the standard approach of showing a lower bound for deterministic algorithms that succeed with a sufficiently large probability, when sampling the input graph from a hard distribution, defined as follows: We first flip a fair coin that determines whether we choose the base graph H or a rewired graph. In the latter case, we sample a rewired graph uniformly at random from the set of all possible rewired graphs $\mathcal{R}$. Observe that the algorithm cannot fail on graph H, since this would result in an error probability of at least $\frac{1}{2}$. Consequently, it is sufficient to show a lower bound under the assumption that the algorithm succeeds on a large fraction of the rewired graphs. We point out that a straightforward generalization of the above proof shows that the argument still holds for deterministic algorithms that succeed on a $(1-\beta)$-fraction of the graphs in $\mathcal{R}$ of the base graph H, for a sufficiently small $\beta > 0$. Thus, similarly to [27], a simple application of Yao's minimax lemma yields the sought message complexity lower bound for randomized Monte Carlo algorithms, and completes the proof of the theorem.

4 An Algorithm for Spanning Tree Verification in the KT_0 CONGEST Model

In this section, we give a message-efficient algorithm that verifies whether the input T is an ST of the network G under the One-Detects assumption, in the setting where all nodes have knowledge of some α-approximation $\tilde{n}$ of the network size n, for some $\alpha < \sqrt{2}$; formally speaking, $\tilde{n} \in [n/\alpha, \alpha n]$. Our result stands in contrast to the strong lower bound of $\Omega(n^2)$ shown by [17] that holds for deterministic ST verification assuming All-Detect and without *any* knowledge of the network size.

We obtain our algorithm by adapting the classic GHS algorithm for constructing an ST, see [10]. However, in contrast to the GHS algorithm, we do not employ the communication-costly operation of exchanging the fragment IDs between neighboring nodes, which requires

$\Omega(m)$ messages per iteration. Considering that our goal is to verify that a given tree T is indeed a spanning tree, we can select an arbitrary edge e from T (when growing a fragment) that is incident to some vertex of the fragment: we can stop the growing process immediately if e turns out to close a cycle. We prove the following result in Appendix B:

▶ **Theorem 19.** *Suppose that all nodes know an α-approximation of the network size, for some $\alpha < \sqrt{2}$. There is a deterministic KT_0 CONGEST algorithm that solves spanning tree verification with a message complexity of $O(n \log n)$ and a time complexity of $O(n \log n)$ rounds while ensuring at least one node detects illegal inputs (i.e., One-Detects). Moreover, if nodes have perfect knowledge of the network size, the algorithm guarantees All-Detect.*

While our main focus is on the message complexity, we point out that the round complexity of the algorithm in Theorem 19 can be as large as $O(n \log n)$. This comes as no surprise, considering that the state-of-the-art solution [23, 15] for computing a spanning tree with $O(n \operatorname{poly} \log n)$ messages (even under the stronger KT_1 assumption) requires at least $\Omega(n)$ rounds.

5 Algorithms for Verifying a d-Approximate BFS Tree

We now turn our attention to the d-BFS tree verification problem. The following lemma suggests that we can extend the algorithm for ST verification described in Section 4 by inspecting the neighborhood of the root when considering a sufficiently large stretch d:

▶ **Lemma 20.** *Let T be a d-approximate BFS tree of G with root r, and suppose that $d \geq \frac{n-1}{x+1}$, for some integer $x \geq 1$. If $\mathsf{dist}_T(r,u) > d \cdot \mathsf{dist}_G(r,u)$ for some node u in G, then $\mathsf{dist}_G(r,u) \leq x$.*

Proof. Consider a node u such that $\mathsf{dist}_T(r,u) > d \cdot \mathsf{dist}_G(r,u)$. It holds that

$$\mathsf{dist}_G(r,u) < \frac{\mathsf{dist}_T(r,u)}{d} \leq \frac{n-1}{d} \leq x+1,$$

and thus $\mathsf{dist}_G(r,u) \leq x$. ◀

▶ **Theorem 21.** *Consider the KT_ρ assumption, for any $\rho \geq 0$. If $d \geq \frac{n-1}{\max\{2,\rho+1\}}$, there exists a deterministic algorithm for d-approximate BFS verification that satisfies One-Detects with a message complexity of $O(n \log n)$ and a time complexity of $O(n \log n)$ rounds, assuming that nodes are given an α-approximation of the network size, for some $\alpha < \sqrt{2}$. If nodes have perfect knowledge of the network size, the algorithm ensures All-Detect.*

We point out that there is no hope of getting $O(n \log n)$ messages for significantly smaller values of d, as the lower bound in Theorem 12 holds for any $d \leq O\left(\frac{n}{4\rho-2}\right)$.

Proof of Theorem 21. First consider the case $\rho \in \{0,1\}$. Instantiating Lemma 20 with $x = 1$, tells us that we only need to check if T is a spanning tree and that all edges incident to the root are in T in order to verify if a subgraph T is a d-approximate BFS. More concretely, after executing the spanning tree verification, each node computes its distance in T from the root. Then the root directly contacts all its neighbors (in G): If there is a node at distance at

least $\frac{n-1}{2}$ from the root (in T) who is not a neighbor of the root, it outputs 0, and broadcasts a `fail` message to all nodes in T; otherwise, it broadcasts an `accept` message. Each node in T decides accordingly once it receives this message from its parent.

The argument for the case $\rho \geq 2$ is similar, except that we now instantiate Lemma 20 with $x = \rho$. That is, since the root knows its ρ-hop neighborhood, it can contact all nodes within distance ρ (in G) by using only $O(n)$ messages. ◀

6 Discussion and Open Problems

In this paper, we study the message complexity of ST verification, MST verification and d-approximate BFS verification distributed algorithms. To the best of our knowledge, the message complexity of d-approximate BFS verification distributed algorithms has never been studied before, hence, we focus our discussion on this problem. In our study, we show that the message complexity is largely determined by the stretch d. When d is small, we obtain a message complexity lower bound of $\Omega(n^2)$ for KT_0 and KT_1, and a lower bound of $\Omega\left(\frac{1}{\rho}\left(\frac{n}{\rho}\right)^{1+\frac{c}{\rho}}\right)$ (for some constant $c > 0$) for KT_ρ where $\rho > 1$. The bound of d is almost tight for KT_0 model, but not for KT_ρ. In particular, for KT_ρ model where $\rho > 1$, it is still open whether we can match the lower bound that holds for $d < \frac{n-1}{\max\{2,\rho+1\}}$.

In addition, all the bounds we obtain for KT_ρ where $\rho \geq 1$ are restricted to comparison-based algorithms, while the bound for KT_0 holds for general algorithms. This gives rise to the following important unanswered question: Can the lower bound of $\Omega(n^2)$ for KT_1 be improved by using non-comparison based algorithms?

References

1. Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In *Distributed Algorithms: 4th International Workshop Bari, Italy, September 24–26, 1990 Proceedings 4*, pages 15–28. Springer, 1991. doi:10.1007/3-540-54099-7_2.
2. Baruch Awerbuch, Amotz Bar-Noy, and Madan Gopal. Approximate distributed bellman-ford algorithms. *IEEE Transactions on Communications*, 42(8):2515–2517, 1994. doi:10.1109/26.310604.
3. Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *Journal of the ACM (JACM)*, 37(2):238–256, 1990. doi:10.1145/77600.77618.
4. Béla Bollobás. *Extremal graph theory*. Courier Corporation, 2004.
5. Brandon Dixon, Monika Rauch, and Robert E Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992. doi:10.1137/0221070.
6. Michael Elkin. Distributed approximation: a survey. *ACM SIGACT News*, 35(4):40–57, 2004. doi:10.1145/1054916.1054931.
7. Laurent Feuilloley, Pierre Fraigniaud, et al. Survey of distributed decision. *Bulletin of EATCS*, 1(119), 2016. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/411.
8. Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *Journal of the ACM (JACM)*, 60(5):1–26, 2013. doi:10.1145/2499228.
9. Greg N Frederickson and Nancy A Lynch. Electing a leader in a synchronous ring. *Journal of the ACM (JACM)*, 34(1):98–115, 1987. doi:10.1145/7531.7919.
10. Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983. doi:10.1145/357195.357200.

11 Dov Harel. A linear algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 185–194, 1985. doi:10.1145/22145.22166.

12 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.

13 Shmuel Katz and Kenneth Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 91–101, 1990. doi:10.1145/93385.93405.

14 Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997. doi:10.1007/BF02526037.

15 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with $o(m)$ communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 71–80, 2015. doi:10.1145/2767386.2767405.

16 Valerie King, Chung Keung Poon, Vijaya Ramachandran, and Santanu Sinha. An optimal erew pram algorithm for minimum spanning tree verification. *Information Processing Letters*, 62(3):153–159, 1997. doi:10.1016/S0020-0190(97)00050-1.

17 Liah Kor, Amos Korman, and David Peleg. Tight bounds for distributed minimum-weight spanning tree verification. *Theory of Computing Systems*, 53(2):318–340, 2013. doi:10.1007/S00224-013-9479-7.

18 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.

19 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.

20 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62(1):7:1–7:27, 2015. doi:10.1145/2699440.

21 Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 381–390, 2013. doi:10.1145/2488608.2488656.

22 Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The journal of logic and algebraic programming*, 78(5):293–303, 2009. doi:10.1016/J.JLAP.2008.08.004.

23 Ali Mashreghi and Valerie King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking, Hyderabad, India, January 5-7, 2017*, page 8. ACM, 2017. URL: http://dl.acm.org/citation.cfm?id=3007775.

24 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.

25 Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1096–115. SIAM, 2020. doi:10.1137/1.9781611975994.67.

26 Shreyas Pai, Gopal Pandurangan, Sriram V. Pemmaraju, Talal Riaz, and Peter Robinson. Symmetry breaking in the congest model: Time- and message-efficient algorithms for ruling sets. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 38:1–38:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.38.

27 Shreyas Pai, Gopal Pandurangan, Sriram V Pemmaraju, and Peter Robinson. Can we break symmetry with $o(m)$ communication? In *PODC'21: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 247–257, 2021. doi:10.1145/3465084.3467909.

28 David Peleg. *Distributed computing: A locality-sensitive approach*. SIAM, 2000.

29 David Peleg and Alejandro A Schäffer. Graph spanners. *Journal of graph theory*, 13(1):99–116, 1989. doi:10.1002/JGT.3190130114.

30 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 Journal on Computing*, 41(5):1235–1265, 2012. doi:10.1137/11085178X.

31 Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the ACM (JACM)*, 26(4):690–715, 1979. doi:10.1145/322154.322161.

A Proof of Lemma 5

▶ **Lemma 5** (restated). *Consider any ϵ-error randomized algorithm $\mathcal{A}$ for a graph verification problem $\mathcal{P}$, where $\epsilon < \frac{1}{6}$. If there exists a hard base graph H for problem $\mathcal{P}$ such that the rewirable components $H[A_1]$ and $H[A_2]$ are both cliques of size $\Theta(n)$, then $\mathcal{A}$ has an expected message complexity of $\Omega(n^2)$ in the* KT_0 LOCAL *model.*

Assume towards a contradiction, that there exists a randomized algorithm $\mathcal{A}$ that satisfies the premise of Lemma 5, while having an expected message complexity of $o(n^2)$. Consider a rewired graph H^{e_1,e_2} where the important edges e_1, e_2 are chosen uniformly at random from their respective rewirable components. Let Few be the event where at most $o(n^2)$ messages are sent and note that the bound on the expected message complexity ensures that Few happens with probability at least $1 - o(1)$. Recall that Found is the event that a message is sent over an important edge (see Lemma 3). We start our analysis by proving an upper bound on the probability that this happens.

▶ **Lemma 22.** $\mathbf{Pr}[\mathsf{Found}] \leq \frac{1}{2}$.

Proof. Since $\mathbf{Pr}[\neg\mathsf{Few}] = o(1)$, we have

$$\mathbf{Pr}[\neg\mathsf{Found}\,] \geq \mathbf{Pr}[\neg\mathsf{Found} \mid \mathsf{Few}]\,\mathbf{Pr}[\mathsf{Few}] \geq \mathbf{Pr}[\neg\mathsf{Found} \mid \mathsf{Few}] - o(1), \tag{2}$$

and hence it will be sufficient to show that $\mathbf{Pr}[\neg\mathsf{Found} \mid \mathsf{Few}] \geq \frac{1}{2} + \Omega(1)$.

We say that a port p incident to some node u is *unexplored*, if u has neither sent nor received a message over p. By assumption, the important edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ of the rewirable graph H are chosen uniformly at random such that $u_1, v_1 \in A_1$ and $u_2, v_2 \in A_2$. Let $n_i = |A_i|$, for $i \in \{1, 2\}$, and recall that $n_i = c_i\, n$, for some constant $c_i > 0$. Conditioned on Few, there must exist a subset $A_1' \subseteq A_1$ of size $(1 - o(1))n_1$, such that every $u \in A_1'$ sends and receives at most $\frac{n_1}{16}$ messages. Since A_1 is a clique and $n_1 = c_1 n$, every such u has at least

$$n_1 - 1 - \frac{n_1}{16} = \frac{15}{16} n_1 - 1 \geq \frac{7}{8} n_1$$

unexplored ports at any point in the execution, for sufficiently large n. A symmetric argument implies the existence of a set $A_2' \subseteq A_2$ with analogous properties. Let A^* be the event that $u_1, v_1 \in A_1'$ and $u_2, v_2 \in A_2'$. We first show that A^* happens with probability at least $1 - o(1)$. The probability that $u_1 \in A_1'$ is $\frac{|A_1'|}{|A_1|} \geq 1 - o(1)$, and analogous statements hold for v_1, u_2, and v_2. Hence, A^* occurs with probability $1 - o(1)$. Now, since $Pr[\neg A^*] = o(1)$, we have

$$\mathbf{Pr}[\neg\mathsf{Found} \mid \mathsf{Few}] \geq \mathbf{Pr}[\neg\mathsf{Found} \mid \mathsf{Few}, A^*] - o(1). \tag{3}$$

Conditioned on Few and A^*, we know that u_1 sends at most $\frac{n_1}{16}$ messages and has at least $\frac{7n_1}{8}$ unexplored ports, and thus the probability that it sends a message over the important edge e_1 is at most $\frac{1}{14}$. Moreover, u_2, v_1, and v_2 send a message over their incident important edge also with probability at most $\frac{1}{14}$. Thus, $\mathbf{Pr}[\neg\mathsf{Found} \mid \mathsf{Few}, A^*]$ is at least $1 - \frac{4}{14} = \frac{5}{7}$. Plugging this into the right-hand side of (3), we obtain

$$\mathbf{Pr}[\neg\mathsf{Found} \mid \mathsf{Few}] \geq \frac{5}{7} - o(1) = \frac{1}{2} + \Omega(1), \tag{4}$$

which is sufficient due to (2). ◀

For a given input $\mathsf{Inp}(G, L, \tilde{n})$ and a subgraph $X \subseteq G$, we define the event $Z[X]$ ("Zero output") as follows:

1. For $\mathsf{One\text{-}Detects}$, $Z[X]$ occurs if at least one node in X outputs 0.
2. For $\mathsf{All\text{-}Detect}$, $Z[X]$ occurs if all nodes in X output 0.

We slightly abuse notation and also write $Z[S]$ when considering the subgraph induced by a set of nodes $S \subseteq V(G)$.

According to Lemma 22, $\mathbf{Pr}(\mathsf{Found}) \leq \frac{1}{2}$, and thus

$$\begin{aligned}&\mathbf{Pr}(Z[H^{e_1,e_2}] \mid \mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n})) \\ &\quad\leq \mathbf{Pr}(Z[H^{e_1,e_2}] \mid \mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n}), \neg\mathsf{Found})\,\mathbf{Pr}(\neg\mathsf{Found}) + \tfrac{1}{2}.\end{aligned} \tag{5}$$

Property (C) of Definition 4 tells us that $\mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n})$ is illegal, and hence the assumption that $\mathcal{A}$ fails with probability at most ϵ implies that

$$\mathbf{Pr}(Z[H^{e_1,e_2}] \mid \mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n})) \geq 1 - \epsilon. \tag{6}$$

Therefore,

$$\begin{aligned}\mathbf{Pr}(Z[H^{e_1,e_2}] \mid \mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n}), \neg\mathsf{Found}) &\geq \mathbf{Pr}(Z[H^{e_1,e_2}] \mid \mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n}), \neg\mathsf{Found}) \\ &\quad \cdot \mathbf{Pr}(\neg\mathsf{Found}) \\ \text{(by (5))} \quad &\geq \mathbf{Pr}(Z[H^{e_1,e_2}] \mid \mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n})) - \tfrac{1}{2} \\ \text{(by (6))} \quad &\geq \tfrac{1}{2} - \epsilon.\end{aligned} \tag{7}$$

Conditioned on $\neg\mathsf{Found}$ (i.e., no important edge is discovered), Lemma 3 tells us that the algorithm behaves the same on the inputs $\mathsf{Inp}(H, L, \tilde{n})$ and $\mathsf{Inp}(H^{e_1,e_2}, L, \tilde{n})$, and hence (7) also yields

$$\mathbf{Pr}(Z[H] \mid \mathsf{Inp}(H, L, \tilde{n})) \geq \tfrac{1}{2} - \epsilon. \tag{8}$$

In Lemma 23 below, we show an upper bound of 2ϵ on the left-hand side of (8), which yields the sought contradiction $\epsilon \geq \frac{1}{6}$, and completes the proof of Lemma 5.

▶ **Lemma 23.** $\mathbf{Pr}(Z[H] \mid \mathsf{Inp}(H, L, \tilde{n})) \leq 2\epsilon$.

Proof. We start by observing that the following inequalities hold for the events $Z[S_1]$ and $Z[S_2]$:

$$\mathbf{Pr}[Z[S_1] \mid \mathsf{Inp}(H[S_1], L[S_1], \tilde{n})] \leq \epsilon. \tag{9}$$

$$\text{If } S_2 \neq \emptyset\text{: } \mathbf{Pr}[Z[S_2] \mid \mathsf{Inp}(H[S_2], L[S_2], \tilde{n})] \leq \epsilon. \tag{10}$$

These inequalities follow from Property (B) of Definition 4, which ensures $\mathsf{Inp}(H[S_1], L[S_1], \tilde{n})$ and $\mathsf{Inp}(H[S_2], L[S_2], \tilde{n})$ are both legal for $\mathcal{P}$. If $S_2 = \emptyset$, then

$$\mathsf{Inp}(H, L, \tilde{n}) = \mathsf{Inp}(H[S_1], L[S_1], \tilde{n}) \leq \epsilon$$

due to (9), and we are done. Thus, we assume that $S_2 \neq \emptyset$ in the remainder of the proof.

First, consider the case that the algorithm satisfies All-Detect, i.e., $Z[H]$ can only be true if $Z[S_1]$ and $Z[S_2]$ are both true: Using the indistinguishability guaranteed by Property (A) of Definition 4, the nodes in S_1 have the same probability distribution over their state transitions in $\mathsf{Inp}(H[S_1], L[S_1], \tilde{n})$ as they do in $\mathsf{Inp}(H, L, \tilde{n})$, and a similar argument applies to the nodes in S_2. From (9), (10), it follows that

$$\begin{aligned}\mathbf{Pr}[Z[H] \mid \mathsf{Inp}(H, L, \tilde{n})] &= \mathbf{Pr}[Z[S_1] \mid \mathsf{Inp}(H[S_1], L[S_1], \tilde{n})]\, \mathbf{Pr}[Z[S_2] | \mathsf{Inp}(H[S_2], L[S_2], \tilde{n})] \\ &\leq \epsilon^2 \leq 2\epsilon.\end{aligned}$$

Now, suppose that the algorithm satisfies One-Detects, where $Z[H]$ is true if either $Z[S_1]$ or $Z[S_2]$ are true. Again, using the indistinguishability guaranteed by Property (A), we obtain that

$$\begin{aligned}\mathbf{Pr}[Z[H] \mid \mathsf{Inp}(H, L, \tilde{n})] &\leq 1 - (1 - \mathbf{Pr}[Z[S_1] \mid \mathsf{Inp}(H[S_1], L[S_1], \tilde{n})]) \\ &\qquad \cdot (1 - \mathbf{Pr}[Z[S_2] \mid \mathsf{Inp}(H[S_2], L[S_2], \tilde{n})]) \\ \text{(by (9) and (10))} \quad &\leq 2\epsilon - \epsilon^2 \leq 2\epsilon.\end{aligned}$$

◀

B Omitted Details of Section 4

In the following, we describe and analyze the deterministic algorithm claimed in Theorem 19. We have omitted some proofs, which can be found in the full paper.

B.1 Description of the Algorithm

Growing Fragments

Each node forms the root of a directed tree, called *fragment*, that initially consists only of itself as the *fragment leader*, and every node in the fragment knows its (current) fragment ID, which is simply the ID of the fragment leader. The algorithm consists of iterations each comprising $c_1\,\tilde{n}$ rounds, for a suitable constant $c_1 \geq 1$, and the goal of an iteration is to merge the fragment with another fragment by finding an *unexplored edge*, i.e., some edge e over which no message has been sent so far. We point out that, in contrast to other ST construction algorithms that follow the Boruvka-style framework of growing fragments, e is *not* guaranteed to lead to another fragment.

In more detail, we proceed as follows: At the start of an iteration, each fragment leader broadcasts along the edges of its fragment F. Upon receiving this message from a parent in F, a node u checks whether it has any incident edges in T over which it has not yet sent a message. If yes, u immediately responds by sending its ID to its parent; otherwise, it forwards the request to all its children in F. If u does not have any children, it immediately sends a nil-response to its parent. On the other hand, if u does have children and it eventually receives a non-nil message from some child, it immediately forwards this response to its parent and ignores all other response messages that it may receive from its other children in this iteration. In the case where u instead received nil responses from every one of its children,

u finally sends a nil message to its own parent. This process ensures that the fragment leader u_ℓ eventually learns the ID of some node v in its fragment that has an unexplored incident tree edge $e_v \in T$, if such a v exists.

Next, u_ℓ sends an `explore!`-message along the tree edges that is forwarded to v, causing v to send a message including the fragment ID (i.e., u_ℓ's ID) on edge e_v, over which it has not yet sent a message. Assume that this edge is connected to some node w.

We distinguish two cases: First, assuming that w is in the same fragment as v, node w responds by sending an `illegal` message to v who upcasts this message to the fragment leader u_ℓ, who, in turn, initiates a downcast of this message to all nodes in the fragment, instructing every node in F to output 0 ("reject"). In this case, the fragment F stops growing and all its nodes terminate. Since we are assuming a synchronous network, it is clear that when nodes from other fragments happen to contact a node from F, they can detect this silence and will also immediately output 0 and terminate.

In the second case, w is in a distinct fragment, and, before we start the next iteration, we run a cycle detection procedure described next.

Acyclicity Check

Since all (non-terminated) fragments attempt to find outgoing edges in parallel in this iteration, we may arrive at the situation where there is a sequence of fragments $F_1, F_2, \ldots, F_k$ such that the unexplored edge found by F_i leads to F_{i+1}, for $i \in [1, k-1]$, and the unexplored edge discovered by F_k may lead "back" to some F_j $(j < k)$. Conceptually, we consider the *fragment graph* $\mathcal{F}$ where vertex f_i corresponds to the fragment leader of F_i, and there is a directed edge from f_i to f_j if the edge explored by F_i points to some node in F_j. (Note that, unlike the case in, e.g.[10], it is not guaranteed that no cycle is formed in the fragment graph, if T is not in fact, a tree.) If two fragments f and f' happen to both explore the same edge e in this iteration, then we say that e is a *core edge*.

▶ **Lemma 24.** *Every component C of $\mathcal{F}$ has at most one cycle. Moreover, C is a tree (i.e., contains no cycle) if and only there exist exactly two fragments f and f' in C that are connected by a core edge.*

Proof. If C contains just a single fragment, the statement is trivial; thus assume that there are at least two fragments in C. Since each fragment explores one outgoing edge, it is clear that C has at most one cycle, which proves the first statement. Now suppose that C is a directed tree. Recalling that each fragment in C has exactly one outgoing edge in $\mathcal{F}$, it follows that there must exist a core edge between two fragments, as otherwise there would be a cycle. Finally, since C has the same number of edges as it has fragments and is connected, it follows that there cannot be any other core edges. ◀

Lemma 24 suggests a simple way for checking whether C contains a cycle. We describe the following operations on $\mathcal{F}$. It is straightforward to translate these operations to the actual network G via broadcasting and convergecasting along the fragment edges. The fragment leaders first confirm with their neighboring fragments (in $\mathcal{F}$) whether they have an incident core edge. Every fragment leader that does not have an incident core edge simply waits by setting a timer of $t = c_2\tilde{n}$ rounds, for a suitable constant $c_2 > 0$. If there exists a core edge between f and f', then the leader with the greater ID, say f, broadcasts a `merge!` message, which is forwarded to all fragments in C by ignoring the direction of the inter-fragment edges, and is guaranteed to arrive within t rounds at every fragment leader of C. Upon receiving this message, every node in C adopts f's ID as its new fragment ID.

On the other hand, if the fragments form a cycle, then, after t rounds, all leaders in C conclude that there is no core edge in C. Lemma 24 ensures that there must be cycle. Thus, the nodes in C do not receive a `merge!` message, causing them to output 0, and terminate at the end of this iteration.

Check Size Requirement

Eventually, in some iteration, it may happen that the fragment leader u_ℓ receives nil messages from all its children, which means that none of the nodes in the fragment has any unexplored edges left. (Note that this also includes the special case where a node does not have any incident edges of T.) In that case, u_ℓ initiates counting the number of nodes in the fragment via a simple broadcast and convergecast mechanism. Once the counting process is complete, the root u_ℓ outputs 0 if the fragment contains less than $\frac{\tilde{n}}{\alpha}$ nodes, and it disseminates its output to all fragment nodes who in turn output 0 and terminate. Otherwise, u_ℓ instructs all nodes to output 1.

On Polynomial Time Local Decision

Eden Aldema Tshuva
Tel Aviv University, Israel

Rotem Oshman
Tel Aviv University, Israel

Abstract

The field of distributed local decision studies the power of local network algorithms, where each network can see only its own local neighborhood, and must act based on this restricted information. Traditionally, the nodes of the network are assumed to have unbounded local computation power, and this makes the model incomparable with centralized notions of efficiency, namely, the classes P and NP. In this work we seek to bridge this gap by studying local algorithms where the nodes are required to be computationally efficient: we introduce the classes PLD and NPLD of polynomial-time local decision and non-deterministic polynomial-time local decision, respectively, and compare them to the centralized complexity classes P and NP, and to the distributed classes LD and NLD, which correspond to local deterministic and non-deterministic decision, respectively.

We show that for deterministic algorithms, requiring both computational and distributed efficiency is likely to be more restrictive than either requirement alone: if the nodes do not know the network size, then $\mathsf{PLD} \subsetneq \mathsf{LD} \cap \mathsf{P}$ holds unconditionally; if the network size is known to all nodes, then the same separation holds under a widely believed complexity assumption ($\mathsf{UP} \cap \mathsf{coUP} \neq \mathsf{P}$). However, when nondeterminism is introduced, this distinction vanishes, and $\mathsf{NPLD} = \mathsf{NLD} \cap \mathsf{NP}$. To complete the picture, we extend the classes PLD and NPLD into a hierarchy akin to the centralized polynomial hierarchy, and we characterize its connections to the centralized polynomial hierarchy and to the distributed local decision hierarchy of Balliu, D'Angelo, Fraigniaud, and Olivetti.

2012 ACM Subject Classification Theory of computation → Complexity classes; Theory of computation → Problems, reductions and completeness

Keywords and phrases Local Decision, Polynomial-Time, LD, NLD

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.27

Funding *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

The field of distributed local decision studies the computation power of distributed network algorithms, where every node can observe only its own neighborhood in the network. There is a rich body of literature characterizing the types of network properties that such algorithms can decide, including deterministic algorithms (e.g., in [23, 7, 6, 8]), randomized algorithms (e.g., in [5, 4]), nondeterministic algorithms (e.g., in [21, 12, 19, 20, 9]), and other variants. However, to our knowledge, all prior work on distributed decision allows the network nodes to have unbounded computation power – they can locally run arbitrary Turing machines (e.g., in [6, 9]), or sometimes even compute any function, even if it is undecidable (e.g., [23]). This puts the theory of efficient distributed decision on different footing from centralized notions of efficiency such as P and NP, and makes the notion of an efficient local distributed algorithm incomparable with that of an efficient centralized algorithm: some problems that are considered hard for a single machine to solve are considered "easy" for a network, simply because we do not take into consideration the computational power of the network nodes. In this paper we introduce a computationally-bounded version of local decision, and study the effects of imposing both locality restrictions and computational efficiency requirements at the same time.

Deterministic computationally bounded local decision. The class LD (for "local decision"), introduced in [9], encompasses all network properties that can be decided by a deterministic constant-round distributed algorithm (see Section 2 for the formal definition). As explained above, this includes algorithms that are not computationally efficient, e.g., algorithms where a network node is required to perform exponential-time computations. Consequently, the class LD is incomparable with classes representing efficient *sequential* computation: it is not contained in, nor does it contain, any class $\mathsf{DTIME}(f(n))$ or $\mathsf{NTIME}(f(n))$ for any time-constructible function $f : \mathbb{N} \to \mathbb{N}$. For instance, there are problems that are known to be NP-hard even in constant-diameter graphs,[1] and these problems are in LD – that is, considered "easy" for local distributed algorithms – even though they would be considered "hard" by mainstream complexity theory.

To study the importance of local computation at the network nodes, we define the class $\mathsf{PLD} \subseteq \mathsf{LD}$, which includes all network properties that can be decided by a deterministic constant-round distributed algorithm that *uses local computation time* $\mathrm{poly}(n)$ *at every node*. The "plain" version of the model does not assume that the size n of the network is known to the nodes, but we also consider a stronger model, $\mathsf{PLD}^{[n]}$, where the nodes do know the network size. We ask:

What is the power of algorithms in PLD / $\mathsf{PLD}^{[n]}$*?*

Clearly, algorithms that are both local and computationally efficient cannot decide languages that are not in LD, nor can they decide languages that are not in P. But can they decide *every language that is both in* LD *and in* P? It turns out that the answer depends on whether or not the network size is known:

- In the "plain" version of the model, where the network size is not known to the nodes, we can prove unconditionally that the answer is *no*: $\mathsf{PLD} \subsetneq \mathsf{LD} \cap \mathsf{P}$.
- In the version where the size is known, we show that the corresponding separation, $\mathsf{PLD}^{[n]} \neq \mathsf{LD}^{[n]} \cap \mathsf{P}$, implies that $\mathsf{P} \neq \mathsf{NP}$. This makes it unlikely that we can prove such a separation unconditionally.
- However, we are able to prove a *conditional* separation: under the assumption that $\mathsf{UP} \cap \mathsf{coUP} \neq \mathsf{P}$, we can show that $\mathsf{PLD}^{[n]} \neq \mathsf{LD}^{[n]} \cap \mathsf{P}$.

The class UP includes all languages that can be decided by a nondeterministic Turing machine with at most one accepting computation path on every input, and the class coUP includes all langauges whose complements are in UP. The intersection $\mathsf{UP} \cap \mathsf{coUP}$ includes the decision version of the problem of integer factorization, which is believed not to be in P, and whose conjectured hardness serves as the basis for RSA public-key encryption.

Our conditional proof that $\mathsf{PLD}^{[n]} \neq \mathsf{LD}^{[n]} \cap \mathsf{P}$ relies on a connection to worst-case cryptography. It is well known that $\mathsf{UP} \cap \mathsf{coUP} \neq \mathsf{P}$ implies the existence of *worst-case one-way functions* [18, 13, 15, 16], that is, functions that can be computed in polynomial time, but cannot be inverted in polynomial time (in the worst case sense: there does not exist a polynomial-time algorithm that correctly computes the inverse of the function on all inputs). Our proof can be viewed as implicitly constructing such a worst-case one-way function from the hardness of $\mathsf{UP} \cap \mathsf{coUP}$, following a construction similar to the one used in [18, 13], and then applying a simple worst-case version of the Goldreich-Levin Theorem [11] to the function we constructed.

Our results for deterministic algorithms are summarized by the following theorem.

[1] For example, the Forwarding Index Problem [25], which asks whether every pair of vertices in the graph can be connected by a path, such that every vertex appears on at most k paths.

▶ **Theorem 1.** *The following holds:*
1. $\mathsf{PLD} \subsetneq \mathsf{P} \cap \mathsf{LD}$.
2. *If* $\mathsf{PLD}^{[n]} \neq \mathsf{P} \cap \mathsf{LD}^{[n]}$, *then* $\mathsf{P} \neq \mathsf{NP}$.
3. *Assuming that* $\mathsf{UP} \cap \mathsf{coUP} \neq \mathsf{P}$, *we have* $\mathsf{PLD}^{[n]} \subsetneq \mathsf{P} \cap \mathsf{LD}^{[n]}$.

Nondeterministic computationally bounded local decision and beyond. In [9], the class NLD (for "nondeterministic local decision") is introduced to capture network properties that can be decided by a nondeterministic constant-round distributed algorithm: this is an algorithm where every node is given a *certificate* (or *witness*). The class NLD can be viewed as a distributed analog of NP. It is of particular importance because of its connections to self-stabilization and fault-tolerance: certificates are used in *proof labeling schemes* [22] to help identify illegal network configurations that require attention.

Just as we did with deterministic algorithms, we introduce a computationally-bounded version of NLD, which we call NPLD, and which captures network properties that can be decided nondeterministically by a constant-round algorithm where nodes run in polynomial local time. We again ask whether the restriction to both distributed and computational efficiency is more restrictive than either restriction alone: is $\mathsf{NPLD} = \mathsf{NLD} \cap \mathsf{NP}$? Perhaps surprisingly, the answer this time is that the combination is *not* more restrictive than either restriction alone:

▶ **Theorem 2.** $\mathsf{NPLD} = \mathsf{NP} \cap \mathsf{NLD}$, *and* $\mathsf{NPLD}^{[n]} = \mathsf{NP} \cap \mathsf{NLD}^{[n]}$.

Here, $\mathsf{NPLD}^{[n]}, \mathsf{NLD}^{[n]}$ are versions of NPLD and NLD (resp.) where the size of the network is known to all nodes. We note that it was shown in [9] that $\mathsf{NLD}^{[n]}$ contains all Turing-decidable languages, which implies that $\mathsf{NP} \cap \mathsf{NLD}^{[n]} = \mathsf{NP}$.

In sequential complexity theory, P and NP are the lowest levels of the *polynomial hierarchy*, $\mathsf{PH} = \{\Sigma_k, \Pi_k\}_{k=0}^{\infty}$, which extends the notion of nondeterminism ("$x \in \mathcal{L}$ iff *there exists* a witness that causes us to accept") and co-nondeterminism ("$x \in \mathcal{L}$ iff *every* witness causes us to accept") into a hierarchy of complexity classes, with an increasing number of quantifier alternations on the witness. At the k-th level of the hierarchy, the class Σ_k allows k quantifier alternations starting with $\exists$, and the class Π_k allows k quantifier alternations starting with $\forall$. The polynomial hierarchy has been the subject of intense research, and one of the most important open problems in complexity theory is whether the inclusions between the levels of the polynomial hierarchy are strict or not. It is commonly believed that they are [10], and "the polynomial hierarchy does not collapse to level k" is a fairly standard hardness assumption, like $\mathsf{P} \neq \mathsf{NP}$.

Inspired by the polynomial hierarchy, in [2], the classes LD and NLD were extended into a full hierarchy of local decision, $\left\{\Sigma_k^{local}, \Pi_k^{local}\right\}_{k=0}^{\infty}$, where the classes in the k-th level of the hierarchy allow k quantifier alternations on the certificates, starting with $\exists$ for Σ and with $\forall$ for Π. The authors of [2] were able to fully characterize the power of each level of the hierarchy: they showed that

$$\mathsf{LD} \subsetneq \Pi_1^{local} \subsetneq \mathsf{NLD} = \Sigma_1^{local} = \Sigma_2^{local} \subsetneq \Pi_2^{local} = \mathsf{ALL}.$$

(The class ALL is defined to be all Turing-decidable languages, and all classes $\Sigma_k^{local}, \Pi_k^{local}$ are restricted to Turing-decidable languages as well.)

What happens to the local decision hierarchy when nodes are restricted to run in polynomial time? We already noted that at the first level, nondeterministic local decision (the class $\Sigma_1^{local} = \Sigma_2^{local} = \mathsf{NLD}$) is unaffected by this additional restriction. We show that the same holds for all levels above as well (Σ_k^{local} for $k \geq 3$, and Π_k^{local} for $k \geq 2$).

Our results for the local decision hierarchy (other than NLD, which was discussed above) are summarized in the following theorems. Here, $\Sigma_k^{\mathsf{P}\text{-}local}$ (resp. $\Pi_k^{\mathsf{P}\text{-}local}$) denotes the class of network properties that can be decided by a Σ_k^{local}-algorithm (resp. Π_k^{local}) where every node runs in poly-time (see Section 2 for the formal definition).

▶ **Theorem 3.** *The following holds:*
1. $\Sigma_2^{\mathsf{P}\text{-}local} = \Sigma_2^{\mathsf{P}} \cap \Sigma_2^{local} \subsetneq \Sigma_2^{\mathsf{P}}$.
2. *For every* $k \geq 2$ *we have* $\Pi_k^{\mathsf{P}\text{-}local} = \Pi_k^{\mathsf{P}}$*, and for every* $k \geq 3$ *we have* $\Sigma_k^{\mathsf{P}\text{-}local} = \Sigma_k^{\mathsf{P}}$.

The relationships between the different levels of the hierarchy $\left\{\Sigma_k^{\mathsf{P}\text{-}local}, \Pi_k^{\mathsf{P}\text{-}local}\right\}_{k=0}^{\infty}$ are as follows:

▶ **Theorem 4.** *The following holds:*
1. $\mathsf{PLD} \subsetneq \Pi_1^{\mathsf{P}\text{-}local} \cap \mathsf{NPLD}$.
2. $\mathsf{NPLD} \not\subseteq \Pi_1^{\mathsf{P}\text{-}local}$.
3. $\mathsf{NPLD} \subsetneq \Pi_2^{\mathsf{P}\text{-}local}$.
4. $\Pi_1^{\mathsf{P}\text{-}local} \subsetneq \Sigma_2^{\mathsf{P}\text{-}local} \cap \Pi_2^{\mathsf{P}\text{-}local}$.
5. *For every* $k \geq 0$*, if either* $\Sigma_{k+1}^{\mathsf{P}\text{-}local}$ *or* $\Pi_{k+1}^{\mathsf{P}\text{-}local}$ *equals* $\Pi_k^{\mathsf{P}\text{-}local}$ *or* $\Sigma_2^{\mathsf{P}\text{-}local}$*, or if* $\Sigma_{k+1}^{\mathsf{P}\text{-}local} = \Pi_{k+1}^{\mathsf{P}\text{-}local}$*, then* $\mathsf{PH} = \Pi_k^{local}$.

A partial and preliminary version of this work appeared as a brief announcement in [1]. While our model allows nodes to run in time that is polynomial in the size of the network, a follow-up work [24] considered a model where the runtime of each node must be polynomial *in the size of its neighborhood*. This yields a different model, where nodes with many neighbors are allowed to use more local computation than nodes with few neighbors.

Organization. For lack of space, many proofs are omitted here; we focus on proving Theorems 1 and 2, in Sections 3 and 4 respectively, omitting some minor technical details. In Section 5 we define the polynomial-time local hierarchy, and sketch the proof of Theorem 3. The remaining proofs will appear in the full version of the paper.

2 Preliminaries

Distributed languages and algorithms. A *distributed language* is a set of *graph configurations* (G, x), where G is an undirected graph and $x : V(G) \to \mathcal{X}$ is an input assignment which assigns an input $x(v)$ from some input domain $\mathcal{X}$ to each node $v \in V(G)$. To simplify the notation, we sometimes write $(G, (x_1, \ldots, x_k))$ or $(G, x_1, \ldots, x_k)$ for a configuration where each node $v \in V(G)$ is given a list of inputs $x_1(v), \ldots, x_k(v)$, assigned by a compound input assignment $x_1 : V(G) \to \mathcal{X}_1, \ldots, x_k : V(G) \to \mathcal{X}_k$.

We assume that nodes have unique identifiers (UIDs) from some fixed UID space $\mathcal{U}$, and that during the execution of a distributed algorithm, each node has access to its own UID and its neighbors' UIDs. We denote by (G, x, id) an *identified graph configuration*, where (G, x) is a graph configuration, and $id : V(G) \to \mathcal{U}$ assigns a UID to each node. Note that the UIDs are not part of the definition of a distributed language. However, they can be used by a distributed algorithm that decides the language, e.g., to break symmetry (see the formal definition of an algorithm below).

We make the standard assumption that in graphs of size n, the inputs and the UIDs assigned to the nodes can be encoded in $\text{poly}(n)$ bits, meaning that an identified configuration with a graph of size n can be represented in $\text{poly}(n)$ bits. (This assumption is not essential, but if the representation length of the inputs and the UIDs is not polynomially-related to the size of the graph, we would need to introduce another parameter to bound their sizes.)

Let $N^t_{G,x,id}(v)$ denotes the t-neighborhood of v in the identified configuration (G, x, id), including the UIDs and the inputs of the nodes in the t-neighborhood. In some cases, when the UID assignment or the input assignment are irrelevant, we may use the notations $N^t_{G,x}(v)$ or $N^t_G(v)$, respectively. When both G and x are clear from the context, we write simply $N^t(v)$. Let $\mathcal{B}^t$ be the set of all t-neighborhoods that appear in some identified graph configuration using inputs from $\mathcal{X}$ and UIDs from $\mathcal{U}$.

Next we formally define local distributed algorithms. In a distributed algorithm, each node observes the neighborhood around itself, and then decides whether to accept or reject:

▶ **Definition 5** (Local decision algorithms). *A* t-local decision algorithm *is a computable mapping* $A : \mathcal{B}^t \to \{0, 1\}$*, which outputs a Boolean value (accept/reject). If* $A(N^t_{G,x,id}(v)) = 1$ *at all nodes* $v \in V(G)$*, then we say that* A accepts (G, x, id)*, and write* $A(G, x, id) = 1$*. We say that* A decides *the distributed language* $\mathcal{L}$ *if for every graph configuration* (G, x) *and for every UID assignment* $id : V(G) \to \mathcal{U}$*,*

$$(G, x) \in \mathcal{L} \qquad \Leftrightarrow \qquad A(G, x, id) = 1.$$

Given a t-local decision algorithm, we refer to t as the algorithm's *locality radius*.

▶ **Definition 6** (The classes LD, PLD). *Let* $t : \mathbb{N} \to \mathbb{N}$ *be a function. A distributed language* $\mathcal{L}$ *is in the class* $\mathsf{LD}(t(n))$ *if it can be decided in graphs of size* n *by a* $t(n)$*-local decision algorithm* A*. We refer to such algorithms as "*$\mathsf{LD}(t)$*-algorithms". If in addition the algorithm* A *(i.e., the mapping from* $t(n)$*-neighborhoods to an accept/reject bit) can be computed by a Turing machine that runs in time* $\mathrm{poly}(n)$ *in graph configurations of size* n*, then* $\mathcal{L}$ *is in the class* $\mathsf{PLD}(t)$*.*

For every function $t : \mathbb{N} \to \mathbb{N}$*, define* $\mathsf{LD}(O(t)) = \bigcup_{c>0} \mathsf{LD}(c \cdot t)$*, and similarly,* $\mathsf{PLD}(O(t)) = \bigcup_{c>0} \mathsf{PLD}(c \cdot t)$*. Let* $\mathsf{LD} = \mathsf{LD}(O(1))$*, and let* $\mathsf{PLD} = \mathsf{PLD}(O(1))$*.*

Note that, as usual in the area of local decision, the local algorithm does not necessarily know the size n of the network; nevertheless, as external observers, we can study the dependence of the algorithm's locality radius and its local running time on n. We introduce a separate class, $\mathsf{PLD}^{[n]}$, for local algorithms where the nodes *do* know the size of the network:

▶ **Definition 7** (The class $\mathsf{PLD}^{[n]}$). *Let* $t : \mathbb{N} \to \mathbb{N}$ *be a function. A distributed language* $\mathcal{L}$ *is in the class* $\mathsf{PLD}^{[n]}(t(n))$ *if the following language* $\mathcal{L}'$ *is in* $\mathsf{PLD}(t(n))$*:*

$$\mathcal{L}' = \{(G, (x, 1^n)) : (G, x) \in \mathcal{L} \text{ and } n = |V(G)|\}.$$

3 Deterministic Polynomial-Time Local Decision

In this section we study deterministic algorithms that are both local and run in polynomial time, and prove Theorem 1.

3.1 Unconditional Separation of PLD from P ∩ LD

We begin by proving the first part of Theorem 1, which states that $\mathsf{PLD} \subsetneq \mathsf{P} \cap \mathsf{LD}$. The non-strict containment, $\mathsf{PLD} \subseteq \mathsf{P} \cap \mathsf{LD}$, is easy to see: every PLD-algorithm is also an LD-algorithm, so $\mathsf{PLD} \subseteq \mathsf{LD}$; moreover, if every node of the network computes its decision in $\mathrm{poly}(n)$ time, then a poly-time centralized Turing machine can simulate the distributed algorithm by computing the output of every node, and accepting iff all nodes accept. Therefore $\mathsf{PLD} \subseteq \mathsf{P}$. The main challenge is to prove that the containment is strict, that is, $\mathsf{PLD} \neq \mathsf{P} \cap \mathsf{LD}$.

High-level overview. To separate PLD from P ∩ LD, we use a variation on the language ITER, which was used in [2] to separate Π_1^{local} from LD with unbounded computation power. We call our variation ITER-BOUND.

The idea is to construct a language of paths, where the center node is given a Turing machine M, two inputs $a, b \in \{0,1\}^*$, and a bound $s \in \mathbb{N}$; the goal is to decide whether M halts on both a and b within at most s computation steps, and accepts either a or b (or both). The bound s may be much larger than the length of the overall size of the configuration: it is encoded in binary. An efficient algorithm cannot afford to run M for s steps and check whether it accepts a or b, but a local algorithm with unbounded computation time can do so, and therefore ITER-BOUND ∈ LD. The bound s serves as a "trap door" that allows unbounded-time local algorithms to decide membership in ITER-BOUND, so that ITER-BOUND ∈ LD.

To make the task solvable for a polynomial-time centralized algorithm we would like to restrict the length n of the path to be at least as long as the number of steps required for M to halt on a and on b. This would allow a centralized algorithm to run the machine M for n steps on a and on b, and check that it halts on both inputs and accepts at least one of them. However, we cannot simply add this restriction on the length of the path, as the resulting language would no longer be in LD: a local algorithm cannot necessarily "see" the entire path and does not know its length. Fortunately, there is a way to indirectly impose this restriction in a way that is locally-checkable: we annotate the nodes of the path (using the inputs of the nodes). On the left side of the path, from the center outwards, we write the sequence of configurations that M goes through in its computation on a, until it halts; on the right side of the path we do the same for b. (In particular, the length of the path must indeed be at least the number of steps required for M to halt on a and on b.) A centralized algorithm can run now either run M for n steps on a and on b as described above, or it can simply examine the computation sequence of M, make sure it obeys the transition function of M, and verify that at either the left or the right side of the path (or both) we have an accepting configuration of M. Thus, ITER-BOUND ∈ P. The annotations can also be checked by a local distributed algorithm which simply verifies that every two neighboring nodes have consecutive computation steps of M, and that nodes at the end of the path (that is, nodes of degree 1) have halting configurations of M. Thus, even after adding the annotations, the language is still in LD.

Finally, we prove that an algorithm that is both local and efficient cannot decide the language ITER-BOUND: intuitively, this is because it can neither afford to run M for s steps, nor can it "see" both endpoints of the path at the same time, to verify that at least one of them has an accepting configuration. The formal proof shows that if there existed a PLD-algorithm for ITER-BOUND then we could use it to decide in polynomial time a language that is not in P.

Detailed construction. Recall that the configuration of a Turing machine consists of the contents of the tape, the location of the tape head, and the current state of the machine. To avoid confusion, in the sequel we refer to configurations of Turing machines as *TM-configurations*, and to graph configurations as simply *configurations*.

Given a Turing machine M, an input $a \in \{0,1\}^*$, and a number $i \in \mathbb{N}$, let $\mathsf{config}(M, a, i)$ denote the Turing machine configuration that M reaches after taking i steps on input a; if M halts in fewer than i steps on input a, then $\mathsf{config}(M, a, i)$ is the configuration in which M halts on input a.

Let M be a Turing machine, let $a, b \in \{0,1\}^*$ be inputs on which M halts, and let $s, n_L, n_R \in \mathbb{N}$ such that $n_L \geq |a|, n_R \geq |b|$ and $s \leq c \cdot 2^{n_L+n_R+1}$, for some constant c whose value will be fixed later.[2] We define a configuration $C^{n_L,n_R}(M,a,b,s) = (G,x)$, as follows:

- G is a path of the form $u_{n_L}, \ldots, u_1, v, w_1, \ldots, w_{n_R}$, consisting of a *pivot node* $v \in V(G)$, a left sub-path $L = u_{n_L}, \ldots, u_1$, and a right sub-path $R = u_1, \ldots, u_{n_R}$.
- The input of the pivot node v is $x(v) = (0, \langle M \rangle, a, b, s)$, where $\langle M \rangle$ is the encoding of the Turing machine M.
- For each node $u_i \in L$ on the left sub-path, the input of u_i is $x(u_i) = (i, \langle M \rangle, \mathsf{config}(M,a,i))$. Similarly, for each node $w_i \in R$ on the right sub-path, the input of w_i is given by $x(w_i) = (i, \langle M \rangle, \mathsf{config}(M,b,i))$.

The language ITER-BOUND consists of all configuration $C^{n_L,n_R}(M,a,b,s)$ such that

- The TM-configurations $\mathsf{config}(M,a,n_L), \mathsf{config}(M,b,n_R)$ that are written at the ends of the two sub-paths are both halting, and
- M halts in at most s steps on a and on b, and accepts at least one of them.

Given a configuration $C^{n_L,n_R}(M,a,b,s) = (G,x)$ as defined above, we say that a node $u \in V(G)$ is *r-central* if the distance of u from the pivot is at most r.

As we explained above, it is not difficult to see that ITER-BOUND can be decided by a local algorithm, and is also in P. In particular, a local algorithm that decides membership in ITER-BOUND only needs to check the following conditions: at any node v,

- If v's input starts with the number 0, then the input is of the form $x(v) = (0, \langle M \rangle, a, b, s)$, where M halts on both a and b in at most s steps, and accepts at least one of them. Also, the degree of v is 2.
- If v's input starts with a number $i > 0$, then the input is of the form $x(v) = (i, \langle M \rangle, \mathsf{cfg})$, and v has a neighbor u whose input is $(i-1, \langle M \rangle, \mathsf{cfg}')$, where cfg' is a TM-configuration that precedes cfg according to the transition function of M. The degree of v must be either 2 or 1. If the degree is 1, then cfg must be a halting configuration of M (that is, the state of M in cfg is either the accepting or the rejecting state of M). If the degree is 2, then v must have another neighbor whose input is $(i+1, \langle M \rangle, \mathsf{cfg}'')$, where cfg'' is the TM-configuration that follows cfg according to the transition function of M.

A centralized poly-time algorithm can decide membership in ITER-BOUND by verifying the same local consistency conditions that the local algorithm checks (e.g., that each non-pivot node's input has a TM-configuration that follows the TM-configuration given to the preceding node, and so on). It can also directly check that at least one of the two endpoints of the path contains an accepting configuration. Finally, to verify that M halts in at most s steps on a (and similarly for b), the centralized algorithm can compute the lengths n_L of the left sub-path, and compare: if $s \geq n_L$, then indeed, since n_L computation steps suffice for M to half on a (which is checked using the local consistency conditions), $s \geq n_L$ steps also suffice. On the other hand, if $s < n_L$, then it is permissible for the algorithm to run M for s steps and check whether it halts – the time required to do so is polynomial in $s < n_L < n$ and in the input size $|a| \leq n_L < n$.

Next we prove that ITER-BOUND is not decidable by a polynomial-time local algorithm. In fact, the claim below generalizes to any sublinear locality radius (that is, the class $\mathsf{PLD}(t(n))$ for any $t(n) = o(n)$), but for the sake of simplicity we state it for a constant locality radius (the class $\mathsf{PLD} = \mathsf{PLD}(O(1))$.

[2] This is to ensure that the size of the graph is indeed polynomially-related to the length of the inputs (in bits). For simplicity, we assume that this constraint is imposed externally, that is, the nodes do not need to verify that their inputs have the correct number of bits. However, it is not difficult to modify the proof to verify that the inputs are not too long, though this requires some modifications to the definition of the language ITER-BOUND.

▷ Claim 8. ITER-BOUND $\notin$ PLD.

Proof. Suppose for the sake of contradiction that there is a PLD-algorithm A that decides ITER-BOUND, and let $t > 0$ be its locality radius. Let $\mathcal{L} \in \mathsf{DTIME}(2^n) \setminus \mathsf{P}$ be some language that is Turing-decidable in time $O(2^n)$ but not in polynomial time, and such that $\epsilon \notin \mathcal{L}$ (here and in the sequel, ϵ denotes the empty word). Such a language exists by the Time Hierarchy Theorem [14]. We claim that using the PLD-algorithm A that decides ITER-BOUND, we can construct a polynomial-time Turing machine that decides $\mathcal{L}$ for inputs of size n for any sufficiently large n, a contradiction to the fact that $\mathcal{L} \notin \mathsf{P}$.

Let M be a $\mathsf{DTIME}(2^n)$-time Turing machine that decides $\mathcal{L}$, and let $f \in O(2^n)$ be a function bounding the running time of M on inputs of length n. We assume that for all $n > 0$ we have $f(n) \geq f(0)$ (if not, simply define $f'(n) = \max(f(n), f(0))$ and use $f'(n)$ in place of $f(n)$).

Given input $z \in \{0,1\}^*$, let $C_z = C^{f(|z|),f(|z|)}(M, \epsilon, z, f(|z|))$ be the configuration that encodes the runs of M on ϵ on the left sub-path, and on z on the right sub-path, until M halts, using sub-paths of length $f(|z|)$ in both directions. Since $f(|z|)$ steps suffice for M to halt on ϵ and on z, but $\epsilon \notin \mathcal{L}$, we have $C_z \in$ ITER-BOUND iff $z \in \mathcal{L}$.

We define a poly-time Turing machine M' that decides $\mathcal{L}$ as follows: on input $z \in \{0,1\}^*$, M' constructs the configuration $C'_z \coloneqq C^{2t,2t}(M, \epsilon, z, f(|z|))$, which is essentially the central portion of C_z, including only $2t$ nodes to the left and to the right of the pivot (a total of $4t + 1$ nodes). Next, M' simulates the local algorithm A at all the nodes of C'_z. Finally, M' accepts iff A outputs 1 at all t-central nodes of C'_z, ignoring the outputs of the other nodes.

It is not difficult to verify that the running time of M' is polynomial in $|z|$, in the description length of M (which is constant), and in t (which is also constant). To show that M' indeed decides $\mathcal{L}$, suppose first that $z \in \mathcal{L}$. Then $C_z \in$ ITER-BOUND by construction, and therefore A must output 1 at all nodes of C_z. But this means that all t-central nodes in C'_z must also accept: for each t-central node u in C'_z, the t-local view of u is the same in C_z and in C'_z, because C'_z is obtained from C_z by removing only nodes at distance greater than t from u. Since the output of u depends only on its t-local view, and we know that u accepts in C_z, it must also accept in C'_z. Therefore M' accepts z.

Now suppose that $z \notin \mathcal{L}$. In this case, $C_z \notin$ ITER-BOUND, because in C_z the two inputs encoded in the configuration are both rejected by M (as $\epsilon, z \notin \mathcal{L}$). We claim that at least one t-central node of C_z must reject; as above, this means that the same node also rejects in C'_z, causing M' to reject z.

Suppose for the sake of contradiction that all t-central nodes of C_z accept. However, since $C_z \notin$ ITER-BOUND, we know that some node of C_z rejects; let u be such a node. The distance of u from the pivot v must be greater than t, since we assumed that no t-central node rejects. Now fix some string $a \in \mathcal{L}$ (which must exist, as $\emptyset \in \mathsf{P}$ and we assumed $\mathcal{L} \notin \mathsf{P}$), let $n' = \max(f(|a|), f(|z|))$, and let $C_{a,z} = C^{n',n'}(M, a, z, f(\max(|a|, |z|)))$ be the configuration encoding the runs of M on a (on the left sub-path) and on z (on the right sub-path), using sub-paths of length n', so that M halts on both by the end of both sub-paths. Since $a \in \mathcal{L}$, we have $C_{a,z} \in$ ITER-BOUND, and thus all nodes must accept $C_{a,z}$. This includes node u. However, u is at distance greater than t from the pivot, and therefore its t-local view is the same in $C_{a,z}$ and in C_z; thus, u also accepts in C_z, a contradiction. ◁

As to the constant c that appears in the definition of the language ITER-BOUND (where we required that $s \leq c \cdot 2^{n_L+n_R+1}$), we can choose c to be any constant such that the function $f \in O(2^n)$ in the proof above satisfies $f(n) \leq c \cdot 2^n$ for sufficiently large n.

3.2 Conditional Separation for Polynomial-Time Local Algorithms with Known Network Size

In the previous section we showed that $\mathsf{P} \cap \mathsf{LD} \not\subseteq \mathsf{PLD}$, but our proof used the fact that the nodes do not know the size of the graph, and therefore their output when the graph is a short path is the same as their output on a long path, provided their local neighborhood stays the same. This is a common assumption in distributed computing, but it could be considered problematic when bounding computational resources as a function of the size of the graph. In classical computational complexity, this issue typically does not arise, as centralized algorithms are able to read their entire input and compute its length – with some notable exceptions, such as sublinear-time algorithms, but there it is usually assumed that the input's size is known to the algorithm.

In this section we ask whether the separation of PLD from $\mathsf{LD} \cap \mathsf{P}$ continues to hold if the size of the network is known: let $\mathsf{LD}^{[n]}, \mathsf{PLD}^{[n]}$ be variants of $\mathsf{LD}, \mathsf{PLD}$ (resp.), where nodes receive the size n of the graph as part of their input. Is it still true that $\mathsf{P} \cap \mathsf{LD}^{[n]} \not\subseteq \mathsf{PLD}^{[n]}$?

In what follows, we prove the second and third parts of Theorem 1. We first prove that $\mathsf{P} \cap \mathsf{LD}^{[n]} \not\subseteq \mathsf{PLD}^{[n]}$ would imply $\mathsf{P} \neq \mathsf{NP}$, which makes an unconditional proof of this separation unlikely. We then show that this separation is implied by the assumption that $\mathsf{UP} \cap \mathsf{coUP} \neq \mathsf{P}$, which is stronger than assuming $\mathsf{P} \neq \mathsf{NP}$, but still reasonable.

▷ Claim 9. If $\mathsf{P} \cap \mathsf{LD}^{[n]} \not\subseteq \mathsf{PLD}^{[n]}$, then $\mathsf{P} \neq \mathsf{NP}$.

Proof. We prove the contrapositive: assume that $\mathsf{P} = \mathsf{NP}$, and let us show that every language $\mathcal{L} \in \mathsf{P} \cap \mathsf{LD}^{[n]}$ is also in $\mathsf{PLD}^{[n]}$.

Let $\mathcal{L} \in \mathsf{P} \cap \mathsf{LD}^{[n]}$. We show that $\mathcal{L} \in \mathsf{PLD}^{[n]}$ by constructing a t-local algorithm for $\mathcal{L}$ where every node runs in polynomial time. The idea is to have each node check whether its t-neighborhood can be extended into a configuration in $\mathcal{L}$. We prove that this decides $\mathcal{L}$, relying on the fact that $\mathcal{L}$ can be decided by a t-local algorithm. Since $\mathcal{L} \in \mathsf{P}$, asking whether there exist a configuration $(G, x) \in \mathcal{L}$ that extends the current t-neighborhood of the node is an NP-question, and since we assumed that $\mathsf{P} = \mathsf{NP}$, the resulting algorithm can also be computed in *deterministic* polynomial time, yielding a $\mathsf{PLD}^{[n]}$-algorithm for $\mathcal{L}$.

More formally, let A be a t-local algorithm where each node accepts its t-neighborhood $N^t(v)$ if and only if there exists an identified configuration $(\tilde{G}, \tilde{x}, \tilde{id}) \in \mathcal{GC}$ such that

- The t-neighborhood of v appears in $(\tilde{G}, \tilde{x}, \tilde{id})$, that is, there exists some $u \in V(\tilde{G})$ such that $N^t(v) = N^t_{\tilde{G},\tilde{x},\tilde{id}}(u)$); and
- $(\tilde{G}, \tilde{x}) \in \mathcal{L}$.

We claim that A decides $\mathcal{L}$, and that A is indeed a $\mathsf{PLD}^{[n]}$-algorithm.

To see that A decides $\mathcal{L}$ we must prove that for every identified configuration (G, x, id),

- If $(G, x) \in \mathcal{L}$ then all nodes accept: indeed, for every node $v \in V(G)$, there exists an identified configuration $(\tilde{G}, \tilde{x}, \tilde{id}) = (G, x, id)$, in which the t-neighborhood of v of course appears, such that $(\tilde{G}, \tilde{x}) = (G, x) \in \mathcal{L}$.
- If all nodes accept, then $(G, x) \in \mathcal{L}$: fix some $\mathsf{LD}^{[n]}$-algorithm B for $\mathcal{L}$, which exists by our assumption that $\mathcal{L} \in \mathsf{LD}^{[n]}$. To show that $(G, x) \in \mathcal{L}$, it suffices to show that under B, all nodes accept in (G, x, id). To that end, let $v \in V(G)$. Because v accepts under A, there exists some identified configuration $(\tilde{G}, \tilde{x}, \tilde{id})$ in which v's t-neighborhood appears, such that $(\tilde{G}, \tilde{x}) \in \mathcal{L}$. But since B decides $\mathcal{L}$, all nodes must accept in $(\tilde{G}, \tilde{x}, \tilde{id})$, which means that v's t-neighborhood is accepted under B.

To see that A is an $\mathsf{PLD}^{[n]}$-algorithm, let M be a poly-time Turing machine that decides $\mathcal{L}$, which exists by our assumption that $\mathcal{L} \in \mathsf{P}$. Observe that A can be implemented by a *nondeterministic* Turing machine M' that takes as input the t-neighborhood of node v, and as witness the identified configuration $(\tilde{G}, \tilde{x}, \tilde{id})$ (verifying first that it is a legal identified

configuration); to verify that the t-neighborhood of v appears in $(\tilde{G}, \tilde{x}, \tilde{id})$, we find the unique node $u \in V(\tilde{G})$ that has $\tilde{id}(u) = id(v)$ and verify that $N^t(v) = N^t_{\tilde{G},\tilde{x},\tilde{id}}(u)$; and to verify that $(\tilde{G}, \tilde{x}) \in \mathcal{L}$, we run M. We note that since the input of node v includes the size of the network in unary (1^n), this computation requires polynomial time in the input of v. Finally, since we assumed that $\mathsf{P} = \mathsf{NP}$, the fact that A can be computed in nondeterministic polynomial time also implies that it can be computed in deterministic polynomial time, as desired. ◁

Next we prove that the separation $\mathsf{PLD} \subsetneq \mathsf{P} \cap \mathsf{LD}$ holds assuming that $\mathsf{UP} \cap \mathsf{coUP} \neq \mathsf{P}$. First, let us formally define the class UP [26], which stands for "unambiguous nondeterministic polynomial-time":

▶ **Definition 10.** *The class* UP *is the set of all languages* $\mathcal{L} \subseteq \{0,1\}^*$ *for which there exists a polynomial-time Turing machine* M *and a polynomial* p*, such that*

$$x \in \mathcal{L} \;\Leftrightarrow\; \exists!\, w \in \{0,1\}^{p(|x|)} :\; M(x,w) = 1.$$

Here, $\exists!$ *is the quantifier for unique existence.*

The class coUP is the class of all languages whose complement is in UP. The intersection $\mathsf{UP} \cap \mathsf{coUP}$ contains some natural problems, such as integer factorization and finding the winner of parity games [3, 17]. It is easy to see that $\mathsf{P} \subseteq \mathsf{UP} \cap \mathsf{coUP} \subseteq \mathsf{NP} \cap \mathsf{coNP}$, and it is widely assumed that $\mathsf{UP} \cap \mathsf{coUP} \neq \mathsf{P}$, as otherwise factoring, upon whose hardness some cryptographic systems rely, is decidable in polynomial time.

High-level overview. To prove the separation, we fix some language $\mathcal{L} \in \mathsf{UP} \cap \mathsf{coUP} \setminus \mathsf{P}$, and UP, coUP machines for it, $M_{\mathcal{L}}, M_{\bar{\mathcal{L}}}$, respectively. We construct a distributed language on paths $\mathcal{P}_{\mathcal{L}}$, where each configuration is of the form $C(z, w, j, b)$, such that:

- $z \in \{0,1\}^*$,
- w is either the unique witness for the statement "$z \in \mathcal{L}$" or the unique witness for the statement "$z \notin \mathcal{L}$", depending on which of the two statements is true (clearly both cannot be true at the same time),
- $j \in [|w|]$ is an index, and
- $b = w_j$ is the j-th bit of the witness w.

In the configuration $C(z, w, j, b)$, we give a string z and index $j \in [|w|]$ to every node of the path except for the first and the last nodes of the path. We give the bit b to the first node of the path, and we give the entire witness w to the last node of the path.

It is not hard to see that $\mathcal{P}_{\mathcal{L}} \in \mathsf{P} \cap \mathsf{LD}^{[\mathsf{n}]}$: since the configuration encodes the witness (it is the input of the last node), a centralized algorithm can decide whether a given path configuration is in the language or not in polynomial time, using the machines $M_{\mathcal{L}}$ and $M_{\bar{\mathcal{L}}}$. A local algorithm with unbounded computation power can also decide membership in $\mathcal{P}_{\mathcal{L}}$ by having each node verify that its input is correctly formatted and matches its neighbors; in addition, the first node obtains z and j from its neighbor, computes the witness w using its unbounded computation time, and verifies that $b = w_j$; the last node computes the witness w and verifies that it matches its input.

Now assume for the sake of contradiction that $\mathsf{PLD}^{[\mathsf{n}]} = \mathsf{P} \cap \mathsf{LD}^{[\mathsf{n}]}$. Then since $\mathcal{P}_{\mathcal{L}} \in \mathsf{P} \cap \mathsf{LD}^{[\mathsf{n}]}$, we also have $\mathcal{P}_{\mathcal{L}} \in \mathsf{PLD}^{[\mathsf{n}]}$, meaning there is a t-local algorithm for $\mathcal{P}_{\mathcal{L}}$ where every node runs in poly-time. We show that a centralized poly-time verifier is able to decide membership in the original language $\mathcal{L}$, by essentially "guessing" the witness bit-by-bit: given input $z \in \{0,1\}^*$, the verifier guesses the j-th bit of the witness w for z by simulating the first t nodes of the configuration $C(z, \vec{0}, j, 0)$: if the first t nodes accept, we guess that $w_j = 0$, and otherwise we guess that $w_j = 1$. Recall that in $\mathcal{P}_{\mathcal{L}}$, the witness w is only supposed to be given to the *last* node on the path. We prove that the first t nodes of the configuration must

"know" whether $w_j = 0$ or $w_j = 1$, even though they do not "see" the witness w at the end of the path, and this can be exploited to compute the j-th bit in poly-time. After computing all bits of the witness w, the verifier simply runs $M_{\mathcal{L}}(z, w)$, and outputs the same.

Detailed construction. Let $z, w \in \{0,1\}^*$ be strings, let $j \in [|w|]$ be an index, and let $b \in \{0,1\}$ be a bit. We define a configuration $C(z, w, j, b) = (G, x)$ as follows:

- G is a path $v_1, \ldots, v_n$, where $n = |z|$.
- The input assignment x is given by $x(v_1) = (1, b)$, $x(v_i) = (i, z, j)$ for every $1 < i < n$, and $x(v_n) = (n, w)$.

Let $\mathcal{L}$ be a language such that $\mathcal{L} \in \mathsf{UP} \cap \mathsf{coUP}$, and let $M_{\mathcal{L}}$ and $M_{\bar{\mathcal{L}}}$ be the respective verifying Turing machines for $\mathcal{L}$ and for $\bar{\mathcal{L}}$. We assume w.l.o.g. that $M_{\mathcal{L}}$ and $M_{\bar{\mathcal{L}}}$ take witnesses of the same polynomial length p; that is, let p be such that for every instance z of size n, there exists a unique witness w such that $|w| = p(n)$ and either $M_{\mathcal{L}}(z, w) = 1$ or $M_{\bar{\mathcal{L}}}(z, w) = 1$ (but not both). For a string $z \in \{0,1\}^n$, let $w(z) \in \{0,1\}^{p(n)}$ be that witness.

The language $\mathcal{P}_{\mathcal{L}}$ consists of all configurations $C(z, w, j, b)$ such that $w = w(z)$ and $w_j = b$.

It is not hard to see that for every $\mathcal{L} \in \mathsf{UP} \cap \mathsf{coUP}$, we have $\mathcal{P}_{\mathcal{L}} \in \mathsf{P} \cap \mathsf{LD}^{[n]}$. We now show that for $\mathcal{L} \in \mathsf{UP} \cap \mathsf{coUP} \backslash \mathsf{P}$ we have $\mathcal{P}_{\mathcal{L}} \notin \mathsf{PLD}^{[n]}$.

▷ Claim 11. Assume $\mathsf{UP} \cap \mathsf{coUP} \neq \mathsf{P}$, and let $\mathcal{L} \in \mathsf{UP} \cap \mathsf{coUP} \backslash \mathsf{P}$. Then $\mathcal{P}_{\mathcal{L}} \notin \mathsf{PLD}^{[n]}$.

Proof. Suppose for the sake of contradiction that $\mathcal{P}_{\mathcal{L}} \in \mathsf{PLD}^{[n]}$, and let A be a t-local polynomial-time algorithm for $\mathcal{P}_{\mathcal{L}}$, for some constant t. Let $\mathsf{time}(A)$ denote the running time of A in each node. Then the following centralized, polynomial-time algorithm B decides $\mathcal{L}$ for all inputs of size $n > 2t$, contradicting the fact that $\mathcal{L} \notin \mathsf{P}$. (Inputs of length $\leq 2t$ can be decided by, e.g., exhaustive tabulation – going through all of them and hard-coding the answer into the Turing machine, since there are only finitely many such inputs.)

Given input z of length $|z| = n > 2t$, for each $j \leq p(n)$, let $a_j \in \{0,1\}$ be the bit computed by the following procedure:

1. Construct the configuration $C(z, \vec{0}, j, 0)$, where $\vec{0}$ is a vector comprising $p(n)$ zeroes.
2. Simulate the execution of A on the first t nodes $v_1, \ldots, v_t$ of $C(z, \vec{0}, j, 0)$.
3. If nodes $v_1, \ldots, v_t$ all accept, set $a_j = 0$, and otherwise set $a_j = 1$.

Finally, let $a = a_1, \ldots, a_{p(n)}$. The centralized algorithm B runs $M_{\mathcal{L}}(z, a)$, and accepts iff $M_{\mathcal{L}}(z, a)$ accepts.

To prove the correctness of our algorithm B, fix $z \in \{0,1\}^*$, and let $w = w(z)$ be the witness corresponding to z. First observe that if $z \notin \mathcal{L}$, then algorithm B rejects: in this case there does not exist any string $a \in \{0,1\}^*$ such that $M_{\mathcal{L}}(z, a)$ accepts, so no matter how a is computed, B will always reject.

Now suppose that $z \in \mathcal{L}$. We claim that the string a computed by B is exactly the witness w such that $M_{\mathcal{L}}(z, w)$ accepts: for each $j \in [p(n)]$,

- If $w_j = 0$, then in the configuration $C(z, \vec{0}, j, 0)$, the first t nodes have the same view as they do in $C(z, w, j, w_j = 0)$. (Recall that we assumed $n > 2t$, and therefore the view of the first t nodes does not include v_n, the only node that is given the full witness w.) By definition, $C(z, w, j, w_j) \in \mathcal{P}_{\mathcal{L}}$, and therefore all nodes must accept; thus, the first t nodes must accept in $C(z, \vec{0}, j, 0)$. This causes us to set $a_j = w_j = 0$.
- If $w_j = 1$, then in the configuration $C(z, w, j, 0)$ some node must reject, as this configuration is not in $\mathcal{P}_{\mathcal{L}}$. Moreover, every node v_k where $k > t$ must accept, because all nodes at distance $> t$ from v_1 have the same view that they would in $C(z, w, j, 1)$, which is in $\mathcal{P}_{\mathcal{L}}$. Therefore it is one of the first t nodes that rejects, causing us to set $a_j = w_j = 1$.

This shows that $w = a$, which means that B accepts z.

Regardless of whether $z \in \mathcal{L}$ or not, the runtime of B is polynomial: simulating the execution of a $\mathsf{PLD}^{[n]}$-algorithm at $t = O(1)$ nodes requires polynomial time, and we repeat this $p(n)$ times to compute all the bits of the witness. ◁

4 Adding Nondeterminism

In this section we characterize the computation power of nondeterministic local algorithms with polynomial time local computation, and show that they can decide any language that can be decided by both a computationally unbounded nondeterministic local algorithm and poly-time nondeterministic Turing machine. This holds regardless of whether the size of the network is known or not.

We begin by formally defining the nondeterministic variants of LD, PLD and $\mathsf{PLD}^{[n]}$. Given a graph G, let $\mathcal{C}(G)$ denote the set of all assignments $c : V(G) \to \{0,1\}^*$. A nondeterministic algorithm is one where the nodes are given "nondeterministic advice", in the form of a *certificate assignment* $c \in \mathcal{C}(G)$:

▶ **Definition 12** (The classes NLD, NPLD, $\mathsf{NLD}^{[n]}$ and $\mathsf{NPLD}^{[n]}$)**.** *A distributed language $\mathcal{L}$ is in the class NLD if there exists an LD-algorithm A such that for every configuration (G,x),*

$$(G,x) \in \mathcal{L} \qquad \Leftrightarrow \qquad \exists c \in \mathcal{C}(G) \forall id \, (A \text{ accepts } (G,(x,c),id)) \, .$$

The class NPLD is defined similarly, but the algorithm A is required to run in polynomial time at every node, and the certificate $c(v)$ is required to have polynomial length at every node v.

The classes $\mathsf{NLD}^{[n]}$, $\mathsf{NPLD}^{[n]}$, are the variants of NLD, NPLD (respectively), where the size of the network is known to all nodes: each node receives 1^n in addition to its usual input.

We emphasize that, following [9], the certificates c are chosen *before* the UID assignment; in other words, the certificates may not depend on the UIDs.

Our proof of the first part of Theorem 2, which asserts that $\mathsf{NPLD} = \mathsf{NP} \cap \mathsf{NLD}$, uses a characterization of the class NLD from [7]: NLD is the class of distributed languages that are *closed under lift*. A configuration (G',x') is a *t-lift* of a configuration (G,x) if there exists a mapping $\phi : V(G') \to V(G)$ such that for every $u \in V(G')$, ϕ induces an input-preserving isomorphism between $N^t_{G,x}(\phi(u))$ and $N^t_{G',x'}(u)$, meaning that for each $v \in V(G')$ we have $x'(v) = x(\phi(v))$. A distributed language $\mathcal{L}$ is *closed under lift* if there exists some $t \geq 0$ such that for every configuration $(G,x) \in \mathcal{L}$, all t-lifts of (G,x) are also in $\mathcal{L}$.

▶ **Lemma 13** ([7])**.** NLD *is the class of all distributed languages closed under lift.*

We are now ready to prove the first part of Theorem 2.

▷ Claim 14. $\mathsf{NPLD} = \mathsf{NP} \cap \mathsf{NLD}$.

Proof. The inclusion $\mathsf{NPLD} \subseteq \mathsf{NP} \cap \mathsf{NLD}$ is easy to see, as an NPLD-algorithm is in particular an NLD-algorithm, and it can also be efficiently simulated by a polynomial-time Turing machine that is given all the nodes' certificates.

To see that $\mathsf{NP} \cap \mathsf{NLD} \subseteq \mathsf{NPLD}$, let $\mathcal{L} \in \mathsf{NP} \cap \mathsf{NLD}$, let A be a t-local algorithm for $\mathcal{L}$, and let M be an NP-verifier for $\mathcal{L}$. We construct the following NPLD-algorithm, B: given a configuration (G,x) on n nodes, we give to each node a certificate $c(v) = (i,(G',x'),w)$, where

- $i \in \{1,\ldots,n\}$ is an index,
- G' and x' represent the configuration (G,x), using $\{1,\ldots,n\}$ as the vertices,
- w is an NP-witness such that M accepts $((G',x'),w)$.

The nodes locally verify that

- Their t-neighborhood in G' is isomorphic to their true neighborhood in G, using the indices provided in the certificates as the isomorphism,
- x' correctly describes their input, again using the index,
- They received the same (G', x') as their neighbors, and finally,
- M accepts $((G', x'), w)$.

The first part of the verification passes if and only if (G, x) is a t-lift of (G', x'). If $(G, x) \in \mathcal{L}$, then the certificates specified above cause all nodes to accept. Conversely, if all nodes accept, then (G', x') is a lift of (G, x), and since $M((G', x'), w)$ accepts, we have $(G', x') \in \mathcal{L}$. This implies that $(G, x) \in \mathcal{L}$ as well, because $\mathcal{L}$ is closed under t-lifts. ◁

Finally, consider the setting where the network size is known, that is, the class $\mathsf{NLD}^{[n]}$. In [9], all Turing-decidable distributed languages are shown to be in $\mathsf{NLD}^{[n]}$: the proof is similar to the proof of Claim 14 above, except that when the size of the graph is known, the only possible lift is the graph itself. Computational efficiency was not taken into consideration in [9].

Using a very similar argument, we can modify the proof of Claim 14 to show that $\mathsf{NPLD}^{[n]} = \mathsf{NP} \cap \mathsf{NLD}^{[n]}$: in the proof of Claim 14, if nodes know the network size and check that the configuration (G', x') described in their certificate has that size, then we are guaranteed (G', x') is the true input configuration, rather than merely a lift of it. Thus, when nodes run M on (G', x') using the witness w and verify that it accepts, they are actually verifying that M accepts the true input configuration, i.e., that the input configuration is in the language $\mathcal{L}$.

5 A Polynomial-Time Local Hierarchy

Inspired by the centralized polynomial hierarchy on one hand, and by the local hierarchy of [2] on the other, we conclude by studying a hierarchy that combines both locality and computational constraints.

5.1 Defining a Polynomial Time Local Hierarchy

In what follows, we let $\mathcal{U}(G)$ denote the set of UID assignments $V(G) \to \mathcal{U}$.

▶ **Definition 15** (The classes Σ_k^{local}, Π_k^{local}, $\Sigma_k^{\mathsf{P}\text{-}local}$, $\Pi_k^{\mathsf{P}\text{-}local}$). *A distributed language $\mathcal{L}$ is in the class Σ_k^{local} (resp., $\Sigma_k^{\mathsf{P}\text{-}local}$) if it can be decided by an* LD*-algorithm (resp.,* PLD*-algorithm) A where the nodes are given certificates $c_1, \ldots, c_k$, quantified as follows:*

$$(G, x) \in \mathcal{L} \Rightarrow \exists c_1 \forall c_2 \ldots Q c_k \in \mathcal{C}(G) \forall id \in \mathcal{U}(G) : \; A(G, x, c_1, \ldots, c_k, id) = 1$$
$$(G, x) \notin \mathcal{L} \Rightarrow \forall c_1 \exists c_2 \ldots Q' c_k \in \mathcal{C}(G) \forall id \in \mathcal{U}(G) : \; A(G, x, c_1, \ldots, c_k, id) = 0,$$

where if k is even then Q is the universal quantifier ($\forall$), and if k is odd then Q is the existential quantifier ($\exists$); in both cases, Q' is the opposite quantifier to Q. In the case of the class $\Sigma_k^{\mathsf{P}\text{-}local}$, the certificates $c_1, \ldots, c_k$ are required to be of polynomial length at every node (that is, for some polynomials $p_1, \ldots, p_k$, the quantifiers range only over certificates of length $p_1(n), \ldots, p_k(n)$, respectively).

The class Π_k^{local} (resp., $\Pi_k^{\mathsf{P}\text{-}local}$) is defined similarly to Σ_k^{local} (resp., Π_k^{local}), with the first quantifier being universal instead of existential.

We remark that as in the nondeterministic case, the certificates may not depend on the UIDs (this is the original definition from [2]). This leads to some complications: for example, in the centralized polynomial hierarchy, $\Sigma_1^{\mathsf{P}} = \mathsf{NP}$ and $\Pi_1^{\mathsf{P}} = \mathsf{coNP}$, and in general, for every $k \in \mathbb{N}$ we have $\Pi_k^{\mathsf{P}} = \mathsf{co}\Sigma_k^{\mathsf{P}}$ (in other words, $\mathcal{L} \in \Pi_k^{\mathsf{P}}$ iff $\overline{\mathcal{L}} \in \Sigma_k^{\mathsf{P}}$). This is not the case with the local hierarchy and with the polynomial-time local hierarchy. For example, the complement of the condition "there exist certificates such that under all UID assignments, all nodes accept", which corresponds to Σ_1^{local}, is *not* the condition "for all certificates and for all UID assignments, all nodes accept", which corresponds to membership in Π_1^{local}. Thus, it is not necessarily the case that $\mathcal{L} \in \Pi_1^{local}$ iff $\overline{\mathcal{L}} \in \Sigma_1^{local}$.

Nevertheless, some characteristics of the centralized polynomial hierarchy do carry over to the local hierarchy, and similarly, to the polynomial local hierarchy: for example, for every $k \geq 0$ we have $\Sigma_k^{local} \subseteq \Pi_{k+1}^{local}$ and $\Pi_k^{local} \subseteq \Sigma_{k+1}^{local}$, as we can simply have the algorithm ignore the $(k+1)$-th level of certificates if desired.

5.2 The Upper Levels of The Hierarchy

We show that on the higher levels of the polynomial local hierarchy, the local computation constraint dominates the locality constraint: each higher-level class of the polynomial local hierarchy is *equal* to the corresponding class of the centralized polynomial hierarchy, with no loss in expressiveness caused by the locality restriction.

▷ Claim 16 (Theorem 3, part 2). For every $k \geq 2$ we have $\Pi_k^{\mathsf{P}\text{-}local} = \Pi_k^{\mathsf{P}}$, and for every $k \geq 3$ we have $\Sigma_k^{\mathsf{P}\text{-}local} = \Sigma_k^{\mathsf{P}}$.

The proof follows almost immediately from an argument used in [2] to prove that Π_2^{local} contains all Turing-decidable languages: we can use the same argument to prove Claim 16, taking care to analyze the running times involved.

In [2], to prove that all decidable languages $\mathcal{L}$ are in Π_2^{local}, the authors construct a local distributed algorithm D which, informally speaking, verifies the following statement: "for every configuration (G', x'), either (G', x') is in $\mathcal{L}$, or (G', x') is not the current configuration". (This statement is logically equivalent to asserting that the current configuration is in $\mathcal{L}$.) To that end, the algorithm D is given two certificates: c_1, which is universally quantified, is a description of a configuration (G', x'), along with indices embedding each node into (G', x'), as in the proof of Claim 14; c_2, which is existentially quantified, can take one of two forms:

- If $(G', x') \in \mathcal{L}$, then $c_2 = \perp$ at all nodes.[3] The nodes can verify by themselves that $(G', x') \in \mathcal{L}$, as they have no computational restrictions, and this is what they do in this case.
- If $(G', x') \notin \mathcal{L}$, then c_2 points out some inconsistency that convinces the nodes that (G', x') is not the current configuration. For example, c_2 may prove that c_1 assigns two nodes the same index, or that there is some node whose true neighborhood does not match G', or that there exist two nodes that received different descriptions of (G', x').

Together, both certificates assert that "every input configuration that is not in $\mathcal{L}$ is not the current input configuration", which implies that the current input configuration *is* in $\mathcal{L}$. Although this is not stated explicitly in [2], the algorithm that verifies the certificates c_1, c_2 runs in time polynomial in the size of the graph and the input to the nodes, *except* when $c_2 = \perp$, in which case the nodes need to run some Turing machine whose running time can be unbounded. This is the part we will replace.

[3] This differs from the description of the algorithm in [2], but it is convenient for the way we will use their algorithm later on.

To obtain Claim 16 from the argument of [2] described above, consider a language $\mathcal{L} \in \Pi_k^{\mathsf{P}}$ where $k \geq 2$ is odd (the proof for even k, and for Σ_k^{P} with $k \geq 3$, is similar). Let M be a Π_k^{P}-Turing machine for $\mathcal{L}$, such that $(G, x) \in \mathcal{L}$ iff $\forall w_1 \exists w_2 \ldots \forall w_k M(G, x, w_1, \ldots, w_k)$ accepts. Let D be the algorithm of [2]. We construct a $\Pi_k^{\mathsf{P}\text{-}local}$-algorithm D', which interprets the certificates $c'_1, \ldots, c'_k$ that it receives as follows:

- $c'_1(v) = (c_1(v), w_1)$, where $c_1(v)$ is the universally-quantified certificate from the algorithm D of [2], and w_1 is the universally-quantified first witness of M.
- $c'_2(v) = (c_2(v), w_2)$, where $c_2(v)$ is the existentially-quantified certificate from the algorithm D of [2], and w_2 is the existentially-quantified second witness of M.
- For all $j > 2$ we interpret $c'_j(v) = w_j$, the corresponding witness of M.

The algorithm D' first verifies that all nodes receive the same witnesses $w_1, \ldots, w_k$, and that if $c_2 = \perp$ at some node, then $c_2 = \perp$ at all nodes. If $c_2 = \perp$ at all nodes, then each node runs M on the configuration (G', x') that it extracts from c_1, using the witnesses $w_1, \ldots, w_k$, and outputs the output of M. On the other hand, if $c_2 \neq \perp$, then the nodes collectively run D on the certificates c_1, c_2, and output the output of D. Correctness is similar to that of the original algorithm D: together, the certificates assert that either c_1 describes a configuration in $\mathcal{L}$, or c_1 describes a configuration that differs from the current configuration. This is equivalent to asserting that the current configuration is in $\mathcal{L}$.

The local running time of D' is equal to that of D when $c_2 \neq \perp$, plus the running time of M. Both are polynomial.

The proof of Claim 16 that we sketched above works for any class in the hierarchy that includes an alternation of the form $\forall\exists$, as this is what we require to use the algorithm from [2] (recall that it was originally intended to prove membership in the class Π_2^{local}, which has exactly this quantifier alternation). Thus, the proof applies to $\Pi_k^{\mathsf{P}\text{-}local}$ for $k \geq 2$, and to $\Sigma_k^{\mathsf{P}\text{-}local}$ for $k \geq 3$. What about $\Sigma_2^{\mathsf{P}\text{-}local}$, which has the quantifier alternation $\exists\forall$? It turns out that this class is indeed different: in the claim below, we prove that $\Sigma_2^{\mathsf{P}\text{-}local} = \Sigma_2^{local} \cap \Sigma_2^{\mathsf{P}} \subsetneq \Sigma_2^{\mathsf{P}}$. This shows that for Σ_2, there *is* a loss in expressive power compared to centralized computation: $\Sigma_2^{\mathsf{P}\text{-}local} \subsetneq \Sigma_2^{\mathsf{P}}$, unlike the classes to which Claim 16 applies.

▷ Claim 17 (Theorem 3, part 1). We have $\Sigma_2^{\mathsf{P}\text{-}local} = \Sigma_2^{\mathsf{P}} \cap \Sigma_2^{local} \subsetneq \Sigma_2^{\mathsf{P}}$.

Proof sketch. The containment $\Sigma_2^{\mathsf{P}\text{-}local} = \Sigma_2^{\mathsf{P}} \cap \Sigma_2^{local}$ is, as usual, easy to see. The other direction follows from the fact, proven in [2], that Σ_2^{local} contains only languages that are closed under lift (in fact, [2] proves that $\Sigma_2^{local} = \mathsf{NLD}$). This means we can apply a proof very similar to that of Claim 14: given a language $\mathcal{L} \in \Sigma_2^{\mathsf{P}} \cap \Sigma_2^{local}$, we can construct a $\Sigma_2^{\mathsf{P}\text{-}local}$-algorithm for $\mathcal{L}$ in the same way that we did in Claim 14, except that the algorithm is now also given a second, universally-quantified witness w', which the nodes feed to the Turing machine M along with the first witness w.

As for the claim that $\Sigma_2^{\mathsf{P}} \cap \Sigma_2^{local} \subsetneq \Sigma_2^{\mathsf{P}}$, it suffices to show that there is some language $\mathcal{L} \in \Sigma_2^{\mathsf{P}} \setminus \Sigma_2^{local}$. One such language is defined in [2]: the language EXTS, "exactly two selected", which includes all graph configurations where every node is given a Boolean input and exactly two nodes have the input 1, is not in Σ_2^{local} [2], but it is easy to see that this language is in P, and hence it is in Σ_2^{P}. ◁

References

1 Eden Aldema Tshuva and Rotem Oshman. Brief announcement: On polynomial-time local decision. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 48–50, 2022. doi:10.1145/3519270.3538463.

2 Alkida Balliu, Gianlorenzo D'Angelo, Pierre Fraigniaud, and Dennis Olivetti. What can be verified locally? *Journal of Computer and System Sciences*, 97:106–120, 2018. doi:10.1016/J.JCSS.2018.05.004.

3 Michael R Fellows and Neal Koblitz. Self-witnessing polynomial-time complexity and prime factorization. *Designs, Codes and Cryptography*, 2(3):231–235, 1992. doi:10.1007/BF00141967.

4 Laurent Feuilloley and Pierre Fraigniaud. Randomized local network computing. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 340–349, 2015. doi:10.1145/2755573.2755596.

5 Pierre Fraigniaud, Mika Göös, Amos Korman, Merav Parter, and David Peleg. Randomized distributed decision. *Distributed Computing*, 27(6):419–434, 2014. doi:10.1007/S00446-014-0211-X.

6 Pierre Fraigniaud, Mika Göös, Amos Korman, and Jukka Suomela. What can be decided locally without identifiers? In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 157–165, New York, NY, USA, 2013. ACM. doi:10.1145/2484239.2484264.

7 Pierre Fraigniaud, Magnús M Halldórsson, and Amos Korman. On the impact of identifiers on local decision. In *International Conference On Principles Of Distributed Systems*, pages 224–238, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-35476-2_16.

8 Pierre Fraigniaud, Juho Hirvonen, and Jukka Suomela. Node labels in local decision. *Theoretical Computer Science*, 751:61–73, 2018. doi:10.1016/J.TCS.2017.01.011.

9 Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *Journal of the ACM (JACM)*, 60(5):1–26, 2013. doi:10.1145/2499228.

10 William I Gasarch. Guest column: The third P=? NP poll. *ACM SIGACT News*, 50(1):38–59, 2019. doi:10.1145/3319627.3319636.

11 O. Goldreich and L. A. Levin. A hard-core predicate for all one-way functions. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, pages 25–32, 1989. doi:10.1145/73007.73010.

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

13 Joachim Grollmann and Alan L Selman. Complexity measures for public-key cryptosystems. *SIAM Journal on Computing*, 17(2):309–335, 1988. doi:10.1137/0217018.

14 Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

15 Lane A Hemaspaandra and Jörg Rothe. Characterizing the existence of one-way permutations. *Theoretical Computer Science*, 244(1-2):257–261, 2000. doi:10.1016/S0304-3975(00)00014-1.

16 Christopher M Homan and Mayur Thakur. One-way permutations and self-witnessing languages. *Journal of Computer and System Sciences*, 67(3):608–622, 2003. doi:10.1016/S0022-0000(03)00068-0.

17 Marcin Jurdziński. Deciding the winner in parity games is in UP∩coUP. *Information Processing Letters*, 68(3):119–124, 1998. doi:10.1016/S0020-0190(98)00150-1.

18 Ker-I Ko. On some natural complete operators. *Theoretical Computer Science*, 37:1–30, 1985. doi:10.1016/0304-3975(85)90085-4.

19 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4), 2007. doi:10.1007/S00446-007-0025-1.

20 Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an mst. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 311–320, 2011. doi:10.1145/1993806.1993866.

21 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.

22 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.

23 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

24 Fabian Reiter. A local perspective on the polynomial hierarchy. *arXiv preprint*, 2023. arXiv:2305.09538.

25 Rachid Saad. Complexity of the forwarding index problem. *SIAM Journal on Discrete Mathematics*, 6(3):418–427, 1993. doi:10.1137/0406033.

26 Leslie G Valiant. Relative complexity of checking and evaluating. *Information processing letters*, 5(1):20–23, 1976. doi:10.1016/0020-0190(76)90097-1.

On Asynchrony, Memory, and Communication: Separations and Landscapes

Paola Flocchini
EECS, University of Ottawa, Canada

Nicola Santoro
School of Computer Science, Carleton University, Ottawa, Canada

Yuichi Sudo
Faculty of Computer and Information Sciences, Hosei University, Tokyo, Japan

Koichi Wada
Faculty of Science and Engineering, Hosei University, Tokyo, Japan

Abstract

Research on distributed computing by a team of identical mobile computational entities, called robots, operating in a Euclidean space in *Look-Compute-Move* (*LCM*) cycles, has recently focused on better understanding how the computational power of robots depends on the interplay between their internal capabilities (i.e., persistent memory, communication), captured by the four standard computational models ($\mathcal{OBLOT}$, $\mathcal{LUMI}$, $\mathcal{FSTA}$, and $\mathcal{FCOM}$) and the conditions imposed by the external environment, controlling the activation of the robots and their synchronization of their activities, perceived and modeled as an adversarial scheduler.

We consider a set of adversarial asynchronous schedulers ranging from the classical *semi-synchronous* (Ssynch) and *fully asynchronous* (Asynch) settings, including schedulers (emerging when studying the atomicity of the combination of operations in the *LCM* cycles) whose adversarial power is in between those two. We ask the question: what is the computational relationship between a model M_1 under adversarial scheduler K_1 ($M_1(K_1)$) and a model M_2 under scheduler K_2 ($M_2(K_2)$)? For example, are the robots in $M_1(K_1)$ more powerful (i.e., they can solve more problems) than those in $M_2(K_2)$?

We answer all these questions by providing, through cross-model analysis, a complete characterization of the computational relationship between the power of the four models of robots under the considered asynchronous schedulers. In this process, we also provide qualified answers to several open questions, including the outstanding one on the proper dominance of Ssynch over Asynch in the case of unrestricted visibility.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Look-Compute-Move, Oblivious mobile robots, Robots with lights, Memory versus Communication, Moving and Computing, Asynchrony

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.28

Related Version *Full Version*: https://arxiv.org/abs/2311.03328

Funding This research was partly supported by NSERC through the Discovery Grant program, by JSPS KAKENHI No. 20H04140, 20KK0232, 20K11685, 21K11748, and by JST FOREST Program JPMJFR226U.

1 Introduction

1.1 Background

Robot Models. Since the seminal work of Suzuki and Yamashita [31], the studies of the computational issues arising in distributed systems of mobile computational entities, called *robots*, operating in a Euclidean space have focused on identifying the minimal assumptions

on *internal* capabilities of the robots (e.g., persistent memory, communication) and *external* conditions of the system (e.g., synchrony, activation scheduler) that allow the entities to perform basic tasks and collectively solve given problems.

Endowed with computational, visibility and motorial capabilities, the robots are anonymous (*i.e.*, indistinguishable from each other), uniform (*i.e.*, run the same algorithm), and disoriented (*i.e.*, they might not agree on a common coordinate system). Modeled as mathematical points in the 2D Euclidean plane in which they can freely move, they operate in *Look-Compute-Move* (*LCM*) cycles. In each cycle, a robot "*Look*s " at its surroundings obtaining (in its current local coordinate system) a snapshot indicating the locations of the other robots. Based on this information, the robot executes its algorithm to "*Compute*" a destination, and then "*Move*s" towards the computed location.

In the (weakest and de facto) standard model, $\mathcal{OBLOT}$, the robots are also oblivious (*i.e.*, they have no persistent memory of the past) and silent (*i.e.*, they have no explicit means of communication). Extensive investigations have been carried out to understand the computational limitations and powers of $\mathcal{OBLOT}$ robots for basic coordination tasks such as Gathering (e.g., [1, 2, 4, 8, 9, 10, 17, 25, 31]), Pattern Formation (e.g., [18, 22, 31, 34, 35]), Flocking (e.g., [7, 23, 30]); see also the monograph [14] for a general account.

The absence of persistent memory and the lack of explicit communication critically restrict the computational capabilities of the $\mathcal{OBLOT}$ robots, and limit the solvability of problems. These limitations are removed, to some extent, in the $\mathcal{LUMI}$ model of *luminous* robots. In this model, each robot is equipped with a constant-bounded amount of persistent[1] memory, called *light*, whose value, called *color*, is visible to all robots. In other words, luminous robots can both remember and communicate, albeit at a very limited level. Since its introduction in [11], the model has been the subject of several investigations focusing on the design of algorithms and the feasibility of problems for $\mathcal{LUMI}$ robots (e.g. [3, 11, 12, 20, 24, 27, 28, 29, 32, 33]; see Chapter 11 of [14] for a recent survey). An important result is that, even if so limited, the simultaneous presence of both persistent memory and communication renders luminous robots strictly more powerful than oblivious robots [11]. This has in turns opened the question on the individual computational power of the two internal capabilities, memory and communication, and motivated the investigations on two sub-models of $\mathcal{LUMI}$: the *finite-state* robots denoted as $\mathcal{FSTA}$, where the robots have a constant-size persistent memory but are silent, and the *finite-communication* robots denoted as $\mathcal{FCOM}$, where robots can communicate a constant number of bits but are oblivious (e.g., see [5, 6, 20, 21, 27, 28]).

A/Synchrony. All these studies in all those models have brought to light the crucial role played by two interrelated *external* factors: the level of synchronization and the activation schedule provided by the system. Like in other types of distributed computing systems, there are two different settings, the synchronous and the asynchronous ones.

In the *synchronous* (also called *semi-synchronous*) (SSYNCH) setting, introduced in [31], time is divided into discrete intervals, called *rounds*. In each round, an arbitrary but nonempty subset of the robots is activated, and they simultaneously perform exactly one *Look-Comp-Move* cycle. The selection of which robots are activated at a given round is made by an adversarial scheduler, constrained only to be fair, i.e., every robot is activated infinitely often. Weaker synchronous adversaries have also been introduced and investigated. The most important and extensively studied is the *fully-synchronous* (FSYNCH) scheduler,

[1] i.e., it is not automatically reset at the end of a cycle.

which activates all the robots in every round. Other interesting synchronous schedulers are RSYNCH, where the sets of robots activated in any two consecutive rounds are restricted to be disjoint, and it studied for its use to model energy-restricted robots [6], as well as the family of *sequential* schedulers (e.g., ROUNDROBIN), where in each round only one robot is activated.

In the *asynchronous* setting (ASYNCH), introduced in [16], there is no common notion of time, each robot is activated independently of the others; it allows for finite but arbitrary delays between the *Look*, *Comp* and *Move* phases, and each movement may take a finite but arbitrary amount of time. The duration of each cycle of a robot, as well as the decision of when a robot is activated, are controlled by an adversarial scheduler, constrained only to be fair, i.e., every robot must be activated infinitely often.

Weaker adversaries are easily identified considering the atomicity of the combination of the *Look*, *Comp* and *Move* stages. In particular, if in every cycle the three operations are executed as a single atomic instantaneous operation, this scheduler we shall call LCM-**atomic**-ASYNCH coincides with SSYNCH. On the other hand, by combining fewer operations, two asynchronous schedulers are identified [27]: LC-**atomic**-ASYNCH, where the *Look* and *Comp* operations are a single atomic operation; and CM-**atomic**-ASYNCH, where the *Comp* and *Move* operations are a single atomic operation.

Of independent interest is the restricted asynchronous adversary unable to schedule the *Look* operation of a robot during the *Move* operation of another. The particular theoretical relevance of this scheduler, called M-**atomic**-ASYNCH [27] derives from the fact that one of the strongest debilitating effects of unrestricted asynchrony is precisely the fact that a robot, when looking, cannot detect if another robot is still or moving.

Separators. Like in other types of distributed systems, understanding the computational difference between (levels of) synchrony and asynchrony has been a primary research focus, first in the $\mathcal{OBLOT}$ model, and subsequently in the others.

Indeed, one of the first results in the field has been the proof that in $\mathcal{OBLOT}$ the simple problem of two robots meeting at the same location, called *Rendezvous*(RDV), is unsolvable under SSYNCH [31] while easily solvable under FSYNCH, implying that fully synchronous $\mathcal{OBLOT}$ robots are strictly more powerful than semi-synchronous ones.

Any problem that, like *Rendezvous*, proves the separation between the computational power of robots in two different settings is said to be a *separator*. The quest has immediately been to determine if there are other problems in $\mathcal{OBLOT}$ separating SSYNCH from FSYNCH (i.e., the extent of their computational difference); no other has been found so far. Clearly more important and pressing has been the question of whether there is any computational difference between synchrony and asynchrony. The quest for a problem separating ASYNCH from SSYNCH has been ongoing for more than two decades. Recently a separator has been found in the special case when the visibility range of the robots is limited [26], leaving the existence of a separator open for the unrestricted case.

The quest for a separator in $\mathcal{OBLOT}$ has been made more pressing since the result that no separation exists between ASYNCH and SSYNCH in the $\mathcal{LUMI}$ model [11]; that is, the presence of a limited form of communication and memory is sufficient to completely overcome the limitations imposed by asynchrony. This result has motivated the investigation of the two submodels of $\mathcal{LUMI}$ where the robots are endowed with only the limited form of persistent memory, $\mathcal{FSTA}$, or of communication, $\mathcal{FCOM}$. While separation between fully synchrony and semi-synchrony has been shown to exist for both submodels [5, 21], the more important question of whether one of them is capable of overcoming asynchrony has not yet been answered; indeed, no separator between SSYNCH and ASYNCH has been found so far for either submodel.

Landscapes. To understand the impact that the factors of persistent memory and communication have on the feasibility of problems, the main investigation tool has been the comparative analysis of the (new and/or existing) results obtained for the same problems under the different four models $\mathcal{OBLOT}, \mathcal{FSTA}, \mathcal{FCOM}, \mathcal{LUMI}$. The same methodological tool can obviously be used also to establish the computational relationships between those models within a spectrum of schedulers, so to identify the relative powers of those schedulers within each model.

Through this type of cross-model analysis, researchers have recently produced a comprehensive characterization of the computational relationship between the four models with respect to the range of synchronous schedulers <FSYNCH, RSYNCH, SSYNCH>. creating a comprehensive map of the *synchronous landscape* for distributed systems of autonomous mobile robots in the four models [5, 21].

With respect to the (more powerful) *asynchronous* adversarial schedulers, ranging from LCM-**atomic**-ASYNCH (i.e., SSYNCH) to ASYNCH, very little is known to date on the computational power of persistent memory and of explicit communication in general, and on the computational relationship between the four models in particular. As mentioned, it is known that in $\mathcal{LUMI}$, robots have in ASYNCH the same computational power as in SSYNCH and that asynchronous luminous robots are strictly more powerful than oblivious synchronous robots [11].

Summarizing, while a comprehensive computational map has existed for the synchronous landscape, only disconnected fragments exist so far of the *asynchronous landscape*.

1.2 Contributions

In this paper, we analyze the computational relationship among the four models $\mathcal{OBLOT}$, $\mathcal{FSTA}$, $\mathcal{FCOM}$ and $\mathcal{LUMI}$, under the range of asynchronous schedulers < LCM-**atomic**-ASYNCH, LC-**atomic**-ASYNCH, CM-**atomic**-ASYNCH, M-**atomic**-ASYNCH, and ASYNCH>, establishing a large variety of results. Through these results, we close several open problems, and create a complete map of the asynchronous landscape for distributed systems of autonomous mobile robots in the four models.

Among our contributions, we prove the existence of a separator between SSYNCH and ASYNCH in the standard $\mathcal{OBLOT}$ model for the unrestricted visibility case by identifying a simple natural problem, `Monotone Line Convergence` (`MLCv`), that separates SSYNCH from ASYNCH for $\mathcal{OBLOT}$ robots. This problem requires two robots to convergence towards each other monotonically (i.e., without ever increasing their distance) on the line connecting them. We prove that this problem, trivially solvable in semi-synchronous systems, is however unsolvable if the system is asynchronous.

Because of this separation in $\mathcal{OBLOT}$ on one hand, and of the known absence of separation in $\mathcal{LUMI}$ on the other, the next immediate question is whether either of $\mathcal{LUMI}$'s specific features (i.e., constant-sized communication and persistent memory) is strong enough alone to overcome asynchrony. In other words, are there separators between SSYNCH and ASYNCH in $\mathcal{FSTA}$? in $\mathcal{FCOM}$? In these regards, we provide a positive answer to both questions, thus proving that both features are needed to overcome asynchrony.

The characterization of the computational relationship between the four models with respect to the range of asynchronous schedulers is *complete*: for any two models, $M_1, M_2 \in \{\mathcal{OBLOT}, \mathcal{FSTA}, \mathcal{FCOM}, \mathcal{LUMI}\}$ and adversarial schedulers $K_1, K_2 \in \{LCM$-**atomic**-ASYNCH, LC-**atomic**-ASYNCH, CM-**atomic**-ASYNCH, M-**atomic**-ASYNCH, ASYNCH$\}$ it is determined whether the computational power of (the robots in) M_1 under K_1 is stronger than, weaker than, equivalent to or orthogonal to (i.e., incomparable with) that of (the robots in) M_2 under K_2.

For example, we prove that for $\mathcal{FSTA}$ (i.e., in presence of only limited internal persistent memory), SSYNCH is computationally more powerful than MOVE-**atomic**-ASYNCH, which in turn is computationally more powerful than ASYNCH. The several orthogonality (i.e., incomparability) results include for example the fact that the combination of asynchrony and limited persistent memory is neither more nor less powerful than the combination of synchrony and obliviousness. Observe that to prove that a model under a specific scheduler is stronger than or orthogonal to another model and scheduler (or same model and a different scheduler, or other model and same scheduler) requires to determine a problem solvable in one setting but not in the other.

Among the equivalence of two models each under a specific scheduler, we have proved that for $\mathcal{FCOM}$ (i.e., in presence of only limited communication): the atomic combination of *Compute* and *Move* does not provide any gain with respect to complete asynchrony; on the other hand, the atomic combination of *Look* and *Compute* completely overcomes asynchrony. The proof of the equivalence has involved designing a *simulation protocol* that allows to correctly execute any protocol for the first model and scheduler into the other model and scheduler.

The resulting asynchronous landscape is shown in Figure 1 where S, A, A_{LC} A_M, and A_{CM} denote SSYNCH, ASYNCH, LC-**atomic**-ASYNCH, M-**atomic**-ASYNCH, and CM-**atomic**-ASYNCH, respectively; a box located higher than another indicates dominance unless they are connected by a dashed line, which denotes orthogonality; equivalence is indicated directly in the boxes.

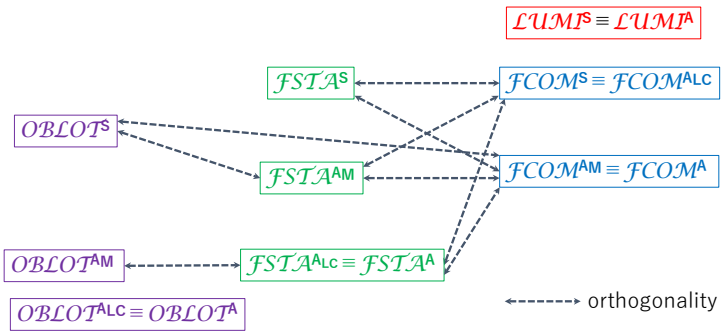


Figure 1 Asynchronous landscape of $\mathcal{LUMI}$, $\mathcal{FCOM}$, $\mathcal{FSTA}$ and $\mathcal{OBLOT}$.

Due to space limitations, some proofs and detailed descriptions are omitted; they can be found in [19].

2 Models and Preliminaries

2.1 Robots

We shall consider a set $R = \{r_0, \cdots, r_{n-1}\}$ of $n > 1$ mobile computational entities, called *robots*, operating in the Euclidean plane $\mathbb{R}^2$. The robots are *anonymous* (i.e., they are indistinguishable by their appearance), *autonomous* (i.e., without central control), *homogeneous* (i.e., the all execute the same program). Viewed as points they can move freely in the plane. Each robot is equipped with a local coordinate system (in which it it is always at its origin),

and it is able to observe the positions of the other robots in its local coordinate system. The robots are *disoriented*; that is, there might not be consistency between the coordinate systems of different robots at the same time, or the same robot at different times[2]. We assume that the robots however have *chirality*; that is, they agree on the the same circular orientation of the plane (e.g., "clockwise" direction).

At any time, a robot is either *active* or *inactive*. When active, a robot r executes a *Look-Compute-Move* (LCM) cycle. Each cycle is compose of three operations:

1. *Look:* The robot obtains an instantaneous snapshot of the positions occupied by the other robots (expressed in its own coordinate system)[3]. We do not assume that the robots are capable of strong multiplicity detection [15].
2. *Compute:* The robot executes its algorithm using the snapshot as input. The result of the computation is a destination point.
3. *Move:* The robot moves to the computed destination[4]; if the destination is the current location, the robot stays still and the move is said to be null.

After executing a cycle, a robot becomes inactive. All robots are initially inactive. The time it takes to complete a cycle is assumed to be finite and the operations *Look* and *Compute* are assumed to be instantaneous.

In the standard model, $\mathcal{OBLOT}$, the robots are also *silent*: they have no explicit means of communication; furthermore, they are *oblivious*: at the start of a cycle, a robot has no memory of observations and computations performed in previous cycles.

In the other common model, $\mathcal{LUMI}$, each robot r is equipped with a persistent register $Light[r]$, called *light*, whose value called *color*, is from a constant-sized set C and is visible by the robots. The color of the light can be set in each cycle by r at the end of its *Compute* operation, and is not automatically reset at the end of a cycle. In $\mathcal{LUMI}$, the *Look* operation produces a colored snapshot; i.e., it returns the set of pairs $(position, color)$ of the other robots. It is sometimes convenient to describe a robot r as having $k \geq 1$ lights, denoted $r.light_1, \ldots, r.light_k$, where the values of $r.light_i$ are from a finite set of colors C_i, and to consider $Light[r]$ as a k-tuple of variables; clearly, this corresponds to r having a single light that uses $\Pi_{i=1}^{k}|C_i|$ colors. Note that if $|C| = 1$, this case corresponds to the $\mathcal{OBLOT}$ model.

Two submodels of $\mathcal{LUMI}$, $\mathcal{FSTA}$ and $\mathcal{FCOM}$, have been defined and investigated, each offering only one of its two capabilities, persistent memory and direct means of communication, respectively. In $\mathcal{FSTA}$, a robot can only see the color of its own light; thus, the color merely encodes an internal state. Therefore, robots are *silent*, as in $\mathcal{OBLOT}$, but they are *finite-state*. In $\mathcal{FCOM}$, a robot can only see the color of the light of the other robots; thus, a robot can communicate to the other robots the color of its light but does not remember its own state (color). Thus, robots are enabled with *finite-communication* but are *oblivious*.

In all the above models, a *configuration* $\mathcal{C}(T)$ at time T is the multiset of the n pairs $(r_i(T), c_i(T))$, where $c_i(T)$ is the color of robot r_i at time T.

---

[2] This is also called *variable* disorientation; restricted forms (e.g., *static* disorientation, where each local coordinate system remains always the same) have been considered for these systems.

[3] This is called the *full visibility* (or unlimited visibility) setting; restricted forms of visibility have also been considered for these systems [17].

[4] This is called the *rigid mobility* setting; restricted forms of mobility (e.g., when movement may be interrupted by an adversary), called *non-rigid mobility* have also been considered for these systems.

2.2 Schedulers, Events

With respect to the activation schedule of the robots, and the duration of their *LCM* cycles, the fundamental distinction is between the *synchronous* and *asynchronous* settings.

In the *synchronous* setting (SSYNCH), also called *semi-synchronous* and first studied in [31], time is divided into discrete intervals, called *rounds*; in each round, a non-empty set of robots is activated and they simultaneously perform a single *Look-Comp-Move* cycle in perfect synchronization. The selection of which robots are activated at a given round is made by an adversarial scheduler, constrained only to be fair (i.e., every robot is activated infinitely often). The particular synchronous setting, where every robot is activated in every round is called *fully-synchronous* (FSYNCH). In a synchronous setting, without loss of generality, the expressions "i-th round" and "time $t = i$" are used as synonyms.

In the *asynchronous* setting (ASYNCH), first studied in [16], there is no common notion of time, the duration of each phase is finite but unpredictable and might be different in different cycles, and each robot is activated independently of the others. The duration of the phases of each cycle as well as the decision of when a robot is activated is controlled by an adversarial scheduler, constrained only to be fair, i.e., every robot must be activated infinitely often.

In the asynchronous settings, the execution by a robot of any of the operations *Look*, *Compute* and *Move* is called an *event*. We associate relevant time information to events: for the *Look* (resp., *Compute*) operation, which is instantaneous, the relevant time is T_L (resp., T_C) when the event occurs; for the *Move* operation, these are the times T_B and T_E when the event begins and ends, respectively. Let $\mathcal{T} = \{T_1, T_2, ...\}$ denote the infinite ordered set of all relevant times; i.e., $T_i < T_{i+1}, i \in \mathbb{N}$. In the following, to simplify the presentation and without any loss of generality, we will refer to $T_i \in \mathcal{T}$ simply by its index i; i.e., the expression "time t" will be used to mean "time T_t".

In our analysis of ASYNCH, we will also consider and make use of the following submodels of ASYNCH, defined by the level of atomicity of the *Look*, *Comp* and *Move* operations.

- ***LC*-atomic-ASYNCH**: The scheduler does not allow any robot r to perform a *Look* operation while another robot $r' \neq r$ is performing its *Comp* operation in that cycle [13, 27]. Thus, in the *LC*-atomic-ASYNCH model, it can be assumed that, in every cycle, the *Look* and *Comp* operations are performed simultaneously and atomically and that $t_L = t_C$.
- ***M*-atomic-ASYNCH**: The scheduler does not allow any robot r to perform a *Look* operation while another robot $r' \neq r$ is performing its *Move* operation in that cycle [13, 27]. In this case, *Move* operations (called M-operations) in all cycles can be considered to be performed instantaneously and that $t_B = t_E$.
- ***CM*-atomic-ASYNCH**: The scheduler does not allow any robot r to perform a *Look* operation while another robot $r' \neq r$ is performing a *Comp* or *Move* operation in that cycle. Thus, in this model, in every cycle the operations *Comp* and *Move*, denoted as CPM, can be considered as performed simultaneously and atomically, and $t_C = t_B = t_E$.

To complete the description, two additional specifications are necessary.

Specification 1. In presence of visible external lights (i.e., models $\mathcal{LUMI}$ and $\mathcal{FCOM}$), if a robot r changes its color in the *Comp* operation at time $t \in \mathcal{T}$, by definition, its new color will become visible only at time $t + 1$.

Specification 2. Under the ***M*-atomic-ASYNCH** and ***CM*-atomic-ASYNCH** schedulers, if a robot r ends a non-null *Move* operation at time $t \in \mathcal{T}$, by definition, its new position will become visible only at time $t + 1$.

Note that, the model where the *Look*, *Comp*, and *Move* operations are considered as a single instantaneous atomic operation (thus referable to as ***LCM*-atomic-**ASYNCH is obviously equivalent to SSYNCH.

In the following, for simplicity of notation, we shall use the symbols F, S, A, A_{LC}, A_M, and A_{CM} to denote the schedulers FSYNCH, SSYNCH, ASYNCH, LC-**atomic**-ASYNCH, M-**atomic**-ASYNCH, and CM-**atomic**-ASYNCH, respectively.

2.3 Problems and Computational Relationships

Let $\mathcal{M} = \{\mathcal{LUMI}, \mathcal{FCOM}, \mathcal{FSTA}, \mathcal{OBLOT}\}$ be the set of models under investigation and $\mathcal{S} = \{F, S, A, A_{LC}, A_M, A_{CM}\}$ be the set of schedulers under consideration.

A problem to be solved (or task to be performed) is described by a set of *temporal geometric predicates*, which implicitly define the *valid* initial, intermediate, and (if existing) terminal[5] configurations, as well as restrictions (if any) on the size n of the set R of robots.

An algorithm $\mathcal{A}$ *solves* a problem P in model $M \in \mathcal{M}$ under scheduler $K \in \mathcal{S}$ if, starting from any valid initial configuration, any execution by R of $\mathcal{A}$ in M under K satisfies the temporal geometric predicates of P.

Given a model $M \in \mathcal{M}$ and a scheduler $K \in \mathcal{S}$, we denote by M^K, the set of problems solvable by robots in M under adversarial scheduler K. Let $M_1, M_2 \in \mathcal{M}$ and $K_1, K_2 \in \mathcal{S}$.

- We say that model M_1 under scheduler K_1 is *computationally not less powerful than* model M_2 under K_2, denoted by $M_1^{K_1} \geq M_2^{K_2}$, if $M_1(K_1) \supseteq M_2(K_2)$.
- We say that M_1 under K_1 is *computationally more powerful than* M_2 under K_2, denoted by $M_1^{K_1} > M_2^{K_2}$, if $M_1^{K_1} \geq M_2^{K_2}$ and $(M_1(K_1) \setminus M_2(K_2)) \neq \emptyset$.
- We say that M_1 under K_1 and M_2 under K_2, are *computationally equivalent* , denoted by $M_1^{K_1} \equiv M_2^{K_2}$, if $M_1^{K_1} \geq M_2^{K_2}$ and $M_2^{K_2} \geq M_1^{K_1}$.
- Finally, we say that K_1 K_2, are *computationally orthogonal* (or *incomparable*), denoted by $M_1^{K_1} \perp M_2^{K_2}$, if $(M_1(K_1) \setminus M_2(K_2)) \neq \emptyset$ and $(M_2(K_2) \setminus M_1(K_1)) \neq \emptyset$.

Trivially,

▶ **Lemma 1.** *For any $M \in \mathcal{M}$ and any $K \in \mathcal{S}$:*

1. $M^F \geq M^S \geq M^{A_{LC}} \geq M^A$
2. $M^F \geq M^S \geq M^{A_{CM}} \geq M^{A_M} \geq M^A$
3. $\mathcal{LUMI}^K \geq \mathcal{FSTA}^K \geq \mathcal{OBLOT}^K$
4. $\mathcal{LUMI}^K \geq \mathcal{FCOM}^K \geq \mathcal{OBLOT}^K$

Let us also recall the following equivalence established in [11]:

▶ **Lemma 2** ([11]). $\mathcal{LUMI}^A \equiv \mathcal{LUMI}^S$

that is, in the $\mathcal{LUMI}$ model, there is no computational difference between ASYNCH and SSYNCH.

Observe that, in all models, any restriction of the adversarial power of the asynchronous scheduler does not decrease (and possibly increases) the computational capabilities of the robots in that model. In other words, if A_α is a restricted scheduler of A_β, then $M^{A_\beta} \leq M^{A_\alpha}$ for any robot model $M \in \mathcal{M}$.

Note that the difference between A_{CM} and A_M is that there exists just one type of configuration that can be observed in A_M but cannot be observed in A_{CM}: the one before moving but after computing. As for $X \in \{\mathcal{FSTA}, \mathcal{OBLOT}\}$, since robots cannot observe the colors of the other robots, we have $X^{A_{CM}} \equiv X^{A_M}$ and $X^{A_{LC}} \equiv X^A$.

[5] A terminal configuration is one in which, once reached, the robots no longer move.

3 The $\mathcal{OBLOT}$ Computational Landscape

3.1 Separating SSYNCH from ASYNCH

In this section we prove that, under SSYNCH, the robots in $\mathcal{OBLOT}$ are strictly more powerful than under $\mathcal{A}_M$, thus separating SSYNCH from ASYNCH in $\mathcal{OBLOT}$.

To do so, we consider the classical `Collisionless Line Convergence` (`CLCv`) problem, where two robots, r and q, must converge to a common location, moving on the line connecting them, without ever crossing each other; i.e., `CLCv` is defined by the predicate

$$\begin{aligned} CLC \equiv \Big[& \{\exists \ell \in \mathbb{R}^2, \forall \epsilon \geq 0, \exists T \geq 0, \forall t \geq T : |r(t) - \ell| + |q(t) - \ell| \leq \epsilon\}, \\ & \textbf{and } \{\forall t \geq 0 : r(t), q(t) \in \overline{r(0)q(0)}\}, \\ & \textbf{and } \{\forall t \geq 0 : dis(r(0), r(t)) \leq dis(r(0), q(t)), dis(q(0), q(t)) \leq dis(q(0), r(t))\}\Big] \end{aligned}$$

and we focus on the monotone version of this problem defined below.

▶ **Definition 3** (MONOTONE LINE CONVERGENCE (`MLCv`)). *The two robots, r and q must solve the* `Collisionless Line Convergence` *problem without ever increasing the distance between them.*

In other words, an algorithm solves `MLCv` iff it satisfies the following predicate:

$$MLC \equiv [CLC \textbf{ and } \{\forall t' \geq t, |r(t') - q(t')| \leq |r(t) - q(t)|\}]$$

First observe that `MLCv` can be solved in $\mathcal{OBLOT}^S$.

▶ **Lemma 4.** `MLCv`$\in \mathcal{OBLOT}^S$. *This holds even under non-rigid movement and in absence of chirality.*

Proof. It is rather immediate to see that the simple protocol using the strategy "move to half distance" satisfies the MLC predicate and thus solves the problem. ◀

On the other hand, `MLCv` is not solvable in $\mathcal{OBLOT}^{\mathcal{A}_M}$.

▶ **Lemma 5.** `MLCv`$\notin \mathcal{OBLOT}^{\mathcal{A}_M}$ *even under fixed disorientation and agreement on the unit of distance.*

Proof. By contradiction, assume that there exists an algorithm $\mathcal{A}$ that solves `MLCv` in $\mathcal{OBLOT}^{\mathcal{A}_M}$. Let the two robots, r and q, have the same unit of distance, initially each see the other on the positive direction of the X axis and their local coordinate system not change during the execution of $\mathcal{A}$. Three observations are in order.

(1) First observe that, by the predicates defining `MLCv`, if a robot moves, it must move towards the other, and in this particular setting, it must stay on its X axis.

(2) Next observe that, every time a robot is activated and executes $\mathcal{A}$, it must move. In fact, if, on the contrary, $\mathcal{A}$ prescribes that a robot activated at some distance d from the other must not move, then, in a fully synchronous execution of $\mathcal{A}$ where both robots are initially at distance d, neither of them will ever move and, thus, will never converge.

(3) Finally observe that, when robot r moves towards q on the X axis after seeing it at distance d, the length $f(d)$ of the computed move is the same as that q would compute if seeing r at distance d.

Consider now the following execution $\mathcal{E}$ under A_M: Initially both robots are simultaneously activated, and are at distance d from each other. Robot r completes its computation and executes the move instantaneously (recall, they are operating under A_M), and continues to be activated and to execute $\mathcal{A}$ while robot q is still in its initial computation.

Each move by r clearly reduces the distance between the two robots. More precisely, by observation (3), after $k \geq 1$ moves, the distance will be reduced from d to d_k where $d_0 = d$ and $d_{k>0} = d_{k-1} - f(d_{k-1}) = d - \sum_{0 \leq i < k} f(d_i)$.

▷ Claim. After a finite number of moves of r, the distance between the two robots becomes smaller that $f(d)$.

Proof. By contradiction, let r never get closer than $f(d)$ to q; that is for every $k > 0$, $d_k > f(d)$.

Consider then the execution $\hat{\mathcal{E}}$ of $\mathcal{A}$ under the RoundRobin synchronous scheduler: the robots, initially at distance d, are activated one per round, at alternate rounds. Observe that, since $\mathcal{A}$ is assumed to be correct under A_M, it must be correct also under RoundRobin. This means that, starting from the initial distance d, for any fixed distance $d' > 0$, the two robots become closer than d'. Let $m(d')$ denote the number of rounds for this to occur; then, the distance between them becomes smaller than $f(d)$ after $m(f(d))$ rounds. Further observe that, after round i, the distance d_i between them is reduced by $f(d_i)$. Summarizing, $d_{m(f(d))} = d - \sum_{0 \leq i < m(f(d))} f(d_i) < f(d)$, contradicting that $d_k > f(d)$ for every $k > 0$. ◁

Consider now the execution $\mathcal{E}$ at the time the distance becomes smaller that $f(d)$; let robot q complete its computation at that time and perform its move, of length $f(d)$, towards r. This move then creates a collision, contradicting the correctness of A. ◀

From Lemmas 4 and 5, and since $\mathcal{OBLOT}^{A_M} \geq \mathcal{OBLOT}^{A}$ by definition, the main result now follows:

▶ **Theorem 6.** $\mathcal{OBLOT}^{S} > \mathcal{OBLOT}^{A}$

In other words, under the synchronous scheduler Ssynch, $\mathcal{OBLOT}$ robots are strictly more powerful than when under the asynchronous scheduler Asynch. This results provides a definite positive answer to the long-open question of whether there exists a computational difference between synchrony and asynchrony in $\mathcal{OBLOT}$.

3.2 Refining the $\mathcal{OBLOT}$ Landscape

We can refine the $\mathcal{OBLOT}$ landscape as follows; By definition, $\mathcal{OBLOT}^{A_M} \geq \mathcal{OBLOT}^{A}$. Consider now the following problem for $n = 4$ robots.

▶ **Definition 7** (TRAPEZOID FORMATION (TF)). *Consider a set of four robots, $R = \{a, b, c, d\}$ whose initial configuration forms a convex quadrilateral $Q = (ABCD) = (a(0)b(0)c(0)d(0))$ with one side, say $\overline{CD}$, longer than all others. The task is to transform Q into a trapezoid T, subject to the following conditions:*

(1) *If Q is a trapezoid, the configuration must stay unchanged (Figure 2(1)); i.e.,*

$$TF1 \equiv [\ Trapezoid(ABCD) \Rightarrow \{\forall t > 0, r \in \{a, b, c, d\} : r(t) = r(0)\}\ ]$$

(2) *Otherwise, without loss of generality, let A be farther than B from CD. Let $Y(A)$ (resp., $Y(B)$) denote the perpendicular lines from A (resp., B) to CD meeting CD in A' (resp. B'), and let α be the smallest angle between $\angle BAA'$ and $\angle ABB'$.*

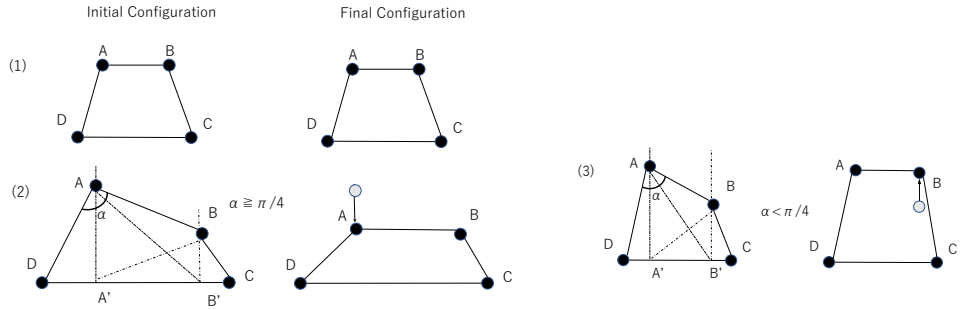


Figure 2 TRAPEZOID FORMATION (TF).

(2.1) *If $\alpha \geq \pi/4$ then the robots must form the trapezoid shown in Figure 2(2), where the location of a is a translation of its initial one on the line $Y(A)$, and that of all other robots is unchanged; specifically,*

$$TF2.1 \equiv [\ (\alpha \geq \pi/4)\ \Rightarrow \{\forall t \geq 0, r \in \{b,c,d\} : r(t) = r(0), a(t) \in Y(A)\} \textbf{ and}$$

$$\{\exists t > 0 : \forall t' \geq t,\ \{\overline{a(t')b(t')}\ ||\overline{CD}\} \textbf{ and } \{a(t') = a(t)\}\ \}\ ]$$

(2.2) *If instead $\alpha < \pi/4$ then the robots must form the trapezoid shown in Fig. 2(3), where the location of all robots but b is unchanged, and that of b is a translation of its initial one on the line $Y(B)$; specifically,*

$$TF2.2 \equiv [\ (\alpha < \pi/4)\ \Rightarrow \{\forall t \geq 0, r \in \{a,c,d\} : r(t) = r(0), b(t) \in Y(B)\} \textbf{ and}$$

$$\{\exists t > 0 : \forall t' \geq t,\ \{\overline{a(t')b(t')}\ ||\overline{CD}\} \textbf{ and } \{b(t') = b(t)\}\ \}\ ]$$

Observe that TF can be solved in $\mathcal{OBLOT}^{A_M}$.

▶ **Lemma 8.** *$TF \in \mathcal{OBLOT}^{A_M}$, even in absence of chirality.*

Proof. It is immediate to see that the following simple set of rules solves TF in $\mathcal{OBLOT}^{A_M}$.

Rule 1: If the observed configuration is as shown in Figure 2 (1), the configuration is already a trapezoid, and no robot performs any move ($TF1$).

Rule 2: Let the configuration be as shown in Figure 2 (2). Whenever observed by b, c, d, none of them moves; when observed by a, a moves to the desired point eventually creating a terminal configuration subject to Rule 1. Since the scheduler is *Move*-atomic ASYNCH the other robots do not observe a during this move, but only after the move is completed.

Rule 3: Analogously, let the configuration be as shown in Figure 2 (3). Whenever observed by a, c, d, none of them moves; when observed by b, b moves to the desired point eventually creating a trapezoid and reaching a terminal configuration, unseen by all other robots during this movement. ◀

However, TF cannot be solved in $\mathcal{OBLOT}^{A}$.

▶ **Lemma 9.** *$TF \notin \mathcal{OBLOT}^{A}$, even with fixed disorientation.*

Proof. By contradiction, let $\mathcal{A}$ be an algorithm that always allows the four $\mathcal{OBLOT}$ robots to solve TF under the asynchronous scheduler. Consider the initial configuration where a is further than b from $\overline{CD}$, and $\alpha = \pi/4$. In this configuration, a is required to move (along $Y(A)$) while no other robot is allowed to move. Observe that, as soon as a moves, it creates a configuration where a is still further than b from $\overline{CD}$, but $\alpha' = \min\{\angle b(t)a(t)A', \angle a(t)b(t)B'\} < \pi/4$.

Consider now the execution of $\mathcal{A}$ in which a is activated first, and then b is activated while a is moving; in this execution, the configuration seen by b requires it to to move, violating $TF2.1$ and contradicting the assumed correctness of algorithm $\mathcal{A}$. ◀

Thus, by Lemmas 8 and 9, we have

▶ **Theorem 10.** $\mathcal{OBLOT}^{\mathcal{A}_M} > \mathcal{OBLOT}^{\mathcal{A}}$

▶ **Theorem 11.** $\mathcal{OBLOT}^{\mathcal{S}} > \mathcal{OBLOT}^{\mathcal{A}_M} > \mathcal{OBLOT}^{\mathcal{A}_{LC}} \equiv \mathcal{OBLOT}^{\mathcal{A}}$

Proof. (1) The equivalence $\mathcal{OBLOT}^{\mathcal{A}_{LC}} \equiv \mathcal{OBLOT}^{\mathcal{A}}$ holds because, by definition, $\mathcal{OBLOT}$ robots cannot distinguish between A_{LC} and A; then, by Theorem 10, $\mathcal{OBLOT}^{\mathcal{A}_M} > \mathcal{OBLOT}^{\mathcal{A}_{LC}}$. (2) It follows from Lemmas 4 and 5. (3) It follows from (1) and Theorem 6. ◀

4 The $\mathcal{FCOM}$ Computational Landscape

4.1 Separating Ssynch from Asynch in $\mathcal{FCOM}$

We have seen (Theorem 6) that, to overcome the limitations imposed by asynchrony, the robots must have some additional power with respect to those held in $\mathcal{OBLOT}$.

In this section, we show that the communication capabilities of $\mathcal{FCOM}$ are not sufficient. In fact, we prove that, under Ssynch, the robots in $\mathcal{FCOM}$ are strictly more powerful than under $\mathcal{A}_M$, thus separating Ssynch from Asynch in $\mathcal{FCOM}$. To do so, we use the problem MLCv again.

Observe that `MLCv` can be solved even in $\mathcal{OBLOT}^{\mathcal{S}}$ (Lemma 4), and thus in $\mathcal{FCOM}^{\mathcal{S}}$.

▶ **Lemma 12.** `MLCv` $\in \mathcal{FCOM}^{\mathcal{S}}$*; this holds even under variable disorientation, non-rigid movement and in absence of chirality.*

On the other hand, `MLCv` is not solvable in $\mathcal{FCOM}^{\mathcal{A}_M}$.

▶ **Lemma 13.** `MLCv` $\notin \mathcal{FCOM}^{\mathcal{A}_M}$.

Proof. Let us consider two robots, r and q, and show that the adversary can activate them in a way that exploits variable disorientation to cause them to violate the condition of `MLCv`.

We consider the execution in which the adversary always forces the robots to perceive the distance between r and q as 1, which is equivalent to the current unit distance of X. We define a function $f(c, d)$ as the length of the move taken by a robot when it observes color c of the other robot and the true distance between the two robots is d in the last Look phase. Since the distance always appears as 1 to the robots, the value $F(c) = f(c, d)/d$ is independent of d. We denote the initial color of the robots as c_0 and assume that $F(c_0) > 0$, which does not affect generality as the adversary can activate r and q multiple times until both robots have a color c such that $F(c) > 0$. Without loss of generality, we also assume that $F(c_0) \leq 1/2$. If $F(c_0) > 1/2$, it follows that r and q pass each other when the adversary activates both robots at time step 0, violating the condition of `MLCv`. Therefore, we assume $0 < F(c_0) < 1/2$ without loss of generality.

Starting from time step 0, the adversary refrains from activating r and instead activates only q to move $\lfloor \log_{1-F(c_0)} F(C_0) \rfloor + 1$ times. Since q always perceives c_0 as the color of r during this period, the distance between r and q decreases by a factor of $(1 - F(C_0))$ with each move of q. As a result, the distance between r and q becomes smaller than $F(C_0)d_0 = f(c_0, d_0)$ after this period, where d_0 is the initial distance between r and q. The adversary then activates r to perform its Move phase. r moves a distance of $f(c_0, d_0)$ and overtakes q, thereby violating the condition. ◀

From Lemmas 12 and 13, and since $\mathcal{FCOM}^{A_M} \geq \mathcal{FCOM}^{A}$ by definition, the main result now follows:

▶ **Theorem 14.** $\mathcal{FCOM}^{S} > \mathcal{FCOM}^{A}$

4.2 Refining the $\mathcal{FCOM}$ Landscape

In this Section we complete the characterization of the asynchronous landscape of $\mathcal{FCOM}$ proving $\mathcal{FCOM}^{A} \equiv \mathcal{FCOM}^{A_M} \equiv \mathcal{FCOM}^{A_{CM}} < \mathcal{FCOM}^{A_{LC}} \equiv \mathcal{FCOM}^{S}$

We first show that every problem solvable by a set of $\mathcal{FCOM}$ robots under A_{CM} can also be solved under ASYNCH. We do so constructively: we present a *simulation* algorithm for $\mathcal{FCOM}$ robots that allows them to correctly execute in ASYNCH any protocol $\mathcal{A}$ given in input (i.e., all its executions under ASYNCH are equivalent to some executions under A_{CM}). The simulation algorithm (called *SIM*) makes each $\mathcal{FCOM}$ robot execute $\mathcal{A}$ infinitely often, never violating the conditions of scheduler CM-**atomic**-ASYNCH. To achieve this, *SIM* needs an activated robot to be able to retrieve some information about its past (e.g., whether or not it has "recently" executed $\mathcal{A}$). Such information can obviously be encoded and persistently stored by the robot in the color of its own light; but, since an $\mathcal{FCOM}$ robot cannot see the color of its light, the robot cannot access the stored information. However, this information can be seen by the other robots, and hence can be communicated by some of them (via the color of their lights) to the needing robot. This can be done efficiently as follows. Exploiting chirality, the robots can agree at any time on a circular ordering of the nodes where robots are located, so that for any such a location x both its predecessor $\texttt{pred}(x)$ and its successor $\texttt{suc}(x)$ in the ordering are uniquely identified, with $\texttt{pred}(\texttt{suc}(x)) = x$; all robots located at x then become responsible for communicating the needed information to the robots located at $\texttt{suc}(x)$[6].

Let $\mathcal{A}$ be an algorithm for $\mathcal{FCOM}$ robots in CM-**atomic**-ASYNCH, and let $\mathcal{A}$ use a light of ℓ colors: $C = \{c_0, c_1, \ldots c_{\ell-1}\}$. It is assumed that, in any initial configuration $\mathcal{C}$, the number of distinct locations[7] is $m \geq 2$.

As described later, the simulation algorithm is composed of four phases. To execute the simulation protocol, a robot r uses four externally visible persistent lights:

1. $r.light \in C$, indicating its own light used in the execution of $\mathcal{A}$; initially, $r.light = c_0$;
2. $r.phase \in \{1, 2, 3, m\}$, indicating the current phase of the simulation algorithm; initially $r.phase = 1$;
3. $r.state \in \{W, M, F\}$, indicating the state of r in its execution of the simulation; initially, $r.state = W$;
4. $r.\texttt{suc}.state \in 2^{\{W,M,F\}}$, indicating the set of states at $\texttt{suc}(\mathrm{x})$, where x is the current location of r; initially, $\mathrm{r}.\texttt{suc}.\mathrm{state}=\{W\}$.

Summarizing, each robot r has $Light[r] = \langle r.light, r.phase, r.state, r.\texttt{suc}.state\rangle$. For a location x and $l \in \{light, phase, state, \texttt{suc}.state\}$, let $x.l = \cup_{\mathrm{r\ at\ x}} r.l$ denote the set of the lights $r.l$ of the robots at location x.

Informally, the simulation algorithm is composed of three main phases which are continuously repeated and a fourth one which is occasionally performed. Each execution of the three main phases corresponds to a single execution of $\mathcal{A}$, each satisfying the CM-atomic

[6] Although we use chirality to determine the cyclic order, this assumption can be circumvented by slightly increasing the number of light colors and deciding the color of the corresponding robot using local 'suc' and 'pred' [14].

[7] In $\mathcal{FCOM}$, by definition, if all robots of the same color are located on the same position, they would not be able to see anything including themselves and they could not perform any task.

condition, by some robots. The three main phases are repeated until every robot has executed $\mathcal{A}$ at least once, ensuring fairness. Appropriate flags are set up to detect this occurrence; a "mega-cycle" is said to be completed, and after the execution of the fourth phase (a reset), a new mega-cycle is started (continuing the simulation of the execution of $\mathcal{A}$ through the continuing execution of the three phases). In other words, in each mega-cycle all robots are activated and execute[8] $\mathcal{A}$ under the CM-**atomic** condition.

Let us describe the protocol in more details. After the *Look* operation, an activated robot r at x recognize its own $r.state$ by using the predecessor's $\texttt{suc}.state$ and the set $r.state.here$ of states seen by r at its own location x. It understands to be in Phase 1 by detecting $\rho.phase = 1$ for any other robot ρ.

In Phase 1, if the activated robot r has not executed yet $\mathcal{A}$ in the current mega-cycle (its state is W) and does not observe any robot with $state = M$, r changes its state flag from W to M and executes $\mathcal{A}$; otherwise, it changes $r.phase$ to 2. Note that, in Phase 1 only a robot with state flag M might be moving; hence since activated robots do not execute $\mathcal{A}$ if they see any robot in phase 1 with state flag M, the simulated algorithm is executed under CM-**atomic**-Asynch. Once all these executions of $\mathcal{A}$ are completed, within finite time every robot r has $r.phase = 2$, and the second phase starts.

In Phase 2, there are no robots executing algorithm $\mathcal{A}$, no robot is moving, and the locations of the robots remains unchanged. An activated robot r at x only updates $r.\texttt{suc}.state$ by observing $\texttt{suc}(x)$ and change its *phase* flag from 2 to 3. When every robot r has $r.phase$ to 3, the third phase starts.

In Phase 3, if robot r executed algorithm $\mathcal{A}$ in Phase 1 ($r.state = M$), then it changes its state flag from M to F (to ensure that the scheduling of robots performing the simulated algorithms is fair). After all robots with M change their state flags to F, every robot copies its neighboring states' flags ($\texttt{suc}(x).state$) setting Phase to 1.

At the beginning of Phase 1, the end of the current mega-cycle is checked. If the mega-cycle is finished (i.e., all robots have their state flags set to F), the robots enter the special Phase m, in which each robot r resets the flag $r.state = W$ and $r.\texttt{suc}.state = \{W\}$. Once completed, the robots return to Phase 1 and begin a new mega-cycle. Observe that, during the transition from Phase 1 to Phase m and from Phase m back to Phase 1, there are configurations containing both phase flags m and 1; it is however not difficult for the robots to distinguish the particular transition being observed: if $\forall r(r.state = F)$is true, it is the former, otherwise $\forall r(r.state = W)$ is true it is the latter.

The correctness of the simulation (see [19]) implies the following:

▶ **Theorem 15.** *$\mathcal{FCOM}^{A_{CM}} \equiv \mathcal{FCOM}^{A}$. This holds even in absence of chirality.*

Since $A_{CM} \leq A_M \leq A$, in turn this theorem implies the following corollary.

▶ **Corollary 16.** *$\mathcal{FCOM}^{A_M} \equiv \mathcal{FCOM}^{A}$. This holds even in absence of chirality.*

We can use the same simulation algorithm to show the equivalence between A_{LC} and Ssynch in $\mathcal{FCOM}$. Since $\mathcal{FCOM}^{A_{LC}} \leq \mathcal{FCOM}^{S}$ by definition, to prove $\mathcal{FCOM}^{A_{LC}} \equiv \mathcal{FCOM}^{A_M}$, we need to show that every problem solvable by a set of $\mathcal{FCOM}$ robots under Ssynch can also be solved under A_{LC}.

The *simulation* algorithm for $\mathcal{FCOM}$ robots that allows them to correctly execute under A_{LC} any protocol designed to work under Ssynch, is actually precisely the simulation algorithm *SIM* described in the previous subsection, that executes under Asynch any

[8] In each phase of mega-cycles, at most one robot may execute the simulated algorithm more than once.

Algorithm 1 `SIM(A)`: predicates and subroutines for robot r at location x.

Assumptions: Let $x_0, x_1, \ldots, x_{m-1}$ be the circular arrangement on the configuration C ($m \geq 2$), and let define $\texttt{suc}(x_i) = x_{i+1 \mod m}$ and $\texttt{pred}(x_i) = x_{i-1 \mod m}$.

State Look
Observe, in particular, $\texttt{pred}(x).state(\texttt{suc}.state)$, $\texttt{suc}(x).state(\texttt{suc}.state)$, and $\rho.phase(\rho \neq r)$;
as well as $r.state.here$ (the set of states seen by r at its own location x.
Note that, for this, r cannot see its own color).

predicate Is-all-phases(p:phase)
$\forall\rho(\neq r)(\rho.phase = p)$

predicate Is-phases-mixed(p, q: phase)
$\forall\rho(\neq r)[(\rho.phase = p)$ **or** $(\rho.phase = q)]$
and [**not** *Is-all-phases(p)*] **and** [**not** *Is-all-phases(q)*]

predicate Is-exist-M
$\exists\rho(\neq r)[(\rho.state = M)$ **or** $(M \in \rho.suc.state)]$

predicate Is-all(s: state)
$\forall\rho(\rho.state = s)$ **and** $own.state = \{s\}$

function r.own.state: set of states
$own.state \leftarrow \texttt{pred}(x).\texttt{suc}.state - x.state.here$,
where $x.state.here$ corresponds to the set of states seen by r at its own location x

subroutine Copy-States-of-Neighbors
r.suc.state $\leftarrow \texttt{suc}(x).state$

subroutine Reset-state-and-neighbor-state
$r.state \leftarrow W$
$r.\texttt{suc}.state \leftarrow \{W\}$

algorithm $\mathcal{A}$ designed to work under A_{CM}. To understand why this is the case, observe that, as we have shown, *any* asynchronous execution of SIM with algorithm $\mathcal{A}$ in input produces a specific execution of $\mathcal{A}$ under A_{CM}. If the executions of SIM were not arbitrary (i.e., under ASYNCH) but under a restricted asynchronous scheduler (say A_α), then each such execution would clearly produce a specific execution of $\mathcal{A}$ under the asynchronous scheduler which satisfies both CM and α.

Thus, the execution of SIM under A_{LC} with $\mathcal{A}$ in input, will produce an execution of $\mathcal{A}$ that satisfies both LC and CM; that is, an execution under LCM-**atomic**-ASYNCH = SSYNCH.

▶ **Theorem 17.** $\mathcal{FCOM}^S \equiv \mathcal{FCOM}^{A_{LC}}$.

Finally, Theorems 14–17 imply the following separation:

▶ **Theorem 18.** $\mathcal{FCOM}^{A_{LC}} > \mathcal{FCOM}^{A_{CM}}$.

Algorithm 2 `SIM(A)` - for robot r at location x.

State Compute

r.des ← r.pos
if *Is-all-phases(1)* **then**
  Copy-state-of-Neighbors
  $r.phase \leftarrow 1$
  if *Is-all(F)* **then** $r.phase \leftarrow m$
  else if $(\exists \rho(\neq r)[(\rho.state = M))$ **then** $r.phase \leftarrow 2$
  else if $(r.own.state = \{W\})$ **then**
    Execute the `Compute` of $\mathcal{A}$ // determining my color *r.light* and destination *r.des* //
    $r.state \leftarrow M$
else if *Is-all-phases(2)* **then**
  $r.phase \leftarrow 3$
  Copy-state-of-Neighbors
else if *Is-all-phases(3)* **then**
  Copy-state-of-Neighbors
  $r.phase \leftarrow 3$
  if *Is-exist-M* **then**
    if $r.own.state = \{M\}$ **then**
      $r.state \leftarrow F$
    Copy-state-of-Neighbors
  else// *no-M*//
    $r.phase \leftarrow 1$
    Copy-state-of-Neighbors
else if *Is-phase-mixed(1,2)* **then**
    $r.phase \leftarrow 2$
else if *Is-phase-mixed(2,3)* **then**
    $r.phase \leftarrow 3$
    Copy-state-of-Neighbors
else if *Is-phase-mixed(1,3)* **then**
    $r.phase \leftarrow 1$
    Copy-state-of-Neighbors
else if *Is-all-phases(m)* **then** //Reset state//
  Reset-state-and-neighbor-state
  if $\exists \rho \neq r(\rho.state = F)$ **then**
    r.phase ← m
  else //There does not exist F//
    r.phase ← 1
else if *Is-phase-mixed(1,m)* **and** *Is-all(F)* **then** $r.phase \leftarrow m$
else if *Is-phase-mixed(1,m)* **and** *Is-all(W)* **then** $r.phase \leftarrow 1$

State Move

Move to *r.des*;

5 The $\mathcal{FSTA}$ Computational Landscape

5.1 Separating SSYNCH from ASYNCH in $\mathcal{FSTA}$

In this section, we consider the $\mathcal{FSTA}$ model; in this model, the only difference with $\mathcal{OBLOT}$ is that the robots are endowed with a bounded amount of memory whose content persists from a cycle to the next. We investigate whether, with this additional capability, the robots are able to overcome the limitations imposed by asynchrony,

The answer is unfortunately negative: we prove that, also in this model, the otherwise enhanced robots are strictly more powerful under the synchronous scheduler SSYNCH than under the asynchronous one ASYNCH.

To do so, we consider the problem MLCv again.

Observe that MLCv can be solved even in $\mathcal{OBLOT}^S$ (Lemma 4), and thus in $\mathcal{FSTA}^S$.

▶ **Lemma 19.** MLCv $\in \mathcal{FSTA}^S$*; this holds even under variable disorientation, non-rigid movement and in absence of chirality.*

On the other hand, MLCv cannot be solved in $\mathcal{FSTA}^{A_M}$.

▶ **Lemma 20.** MLCv $\notin \mathcal{FSTA}^{A_M}$

Proof. Let r and q be the two robots that we consider. In what follows, we will show that for any algorithm, the adversary can activate r and q and exploit variable disorientation so that they violate the condition of MLCv.

Because of variable disorientation, whenever a robot $X \in \{r, q\}$ performs a Look operation, the adversary can (and will) force the *observed* distance between r and q in the resulting snapshot to be always 1 (i.e., equals the current unit distance of X). Let $f(c, d)$ be the length of the computed move when a robot has color c and the *real* distance between the two robots is d in the last Look phase. Note that $F(c) = f(c, d)/d$ does not depend on d because the distance always looks one to the robots.

Since the distance always looks the same to the robots, unless the two robots meet, the transition sequence of the internal colors set by a robot is fixed. In particular, since the number of colors is a fixed constant, after a finite transient, say $(c_0, c_1, \ldots, c_k)$, the sequence becomes periodic, say $(c_s, c_{s+1}, ..., c_k)^*$.

Then, the adversary can activate the robots in the following way so that, either during the transient they violate the condition of MLCv, or both of them end the transient without meeting each other and have color c_s:

1. $i \leftarrow 0$.
2. If $F(c_i) > 1/2$, the adversary activates both r and q, by which they pass each other, clearly violating the condition of MLCv. If $F(c_i) \leq 1/2$, the adversary first activates r, and then activates q, by which r and q never meets (i.e., never reach the same location).
3. $i \leftarrow i + 1$ and go back to 2.

If the robots did not violate the condition of MLCv during their transient, they are both at the beginning of their periodic sequence with color c_s in distinct positions. If $F(c_i) = 0$ holds for all $i = s, s + 1, \ldots, k$, no robot moves, thus MLCv is never solved. So, without loss of generality, we assume $c_s = c_0$ and $F(c_0) > 0$. Then, the following strategy of the adversary leads to the violation of the condition of MLCv, where d_0 is the distance between r and q at time 0.

1. Let r and q perform *Look* and *Compute* phase, by which both r and q compute to move by distance $f(c_0, d_0)$.
2. While r is still waiting to be activated to move, activate only q repeatedly until q overtakes r or the distance between r and q becomes less than $f(c_0, d_0)$. The former case occurs if $F(c_i) > 1$ for some i. This obviously violates the condition of MLCv. Otherwise, the latter case must eventually occur because the distance between r and q becomes constant times smaller each time q changes its color k times. Then, the adversary finally activate r to perform its Move phase. Then, r moves a distance $f(c_0, d_0)$ and overtakes q, violating the condition.

Thus, for any algorithm, the two robots must violate the condition of MLCv. ◀

Thus, by Lemmas 19 and 20, a separation between SSYNCH and ASYNCH in $\mathcal{FSTA}$ is shown.

▶ **Theorem 21.** $\mathcal{FSTA}^S > \mathcal{FSTA}^A$

5.2 Refining the $\mathcal{FSTA}$ landscape

We can refine the $\mathcal{FSTA}$ landscape as follows; Consider again the TF problem defined and analyzed in Section 3.2. By Lemma 8, TF can be solved in $\mathcal{OBLOT}^{A_M}$, and thus in $\mathcal{FSTA}^{A_M}$.

On the other hand, TF is not solvable in $\mathcal{FSTA}^{A_{LC}}$.

▶ **Lemma 22.** *$TF \notin \mathcal{FSTA}^{A_{LC}}$, even with fixed disorientation.*

Proof. By contradiction, let $\mathcal{A}$ be an algorithm that always allows the two $\mathcal{FSTA}$ robots to solve TF and form a trapezoid reaching a terminal state in finite time under the A_{LC} scheduler. Consider the initial configuration where a is further than b from $\overline{CD}$, and $\alpha = \pi/4$. Starting from this configuration, a is required to move within finite time along $Y(A)$; on the other hand, no other robot is allowed to move. Consider now the execution of $\mathcal{A}$ in which only a is activated, and starts moving at time t; observe that, as soon as a moves, it creates a configuration where a is still further than b from $\overline{CD}$, but $\alpha' = \min\{\angle b(t)a(t)A', \angle a(t)b(t)B'\} < \pi/4$.

Activate now b at time $t' > t$ while a is still moving. Should this have been an initial configuration, within a constant number of activations (bounded by the number of internal states), b would move, say at time t". In the current execution, slow down the movement of a so that it is still moving at time t''. Since in $\mathcal{FSTA}$ b cannot access the internal state of a, nor remember previously observed angles and distances, it cannot detect that the observed configurations are not initial configurations; hence it will move at time t", violating $TF2$ and contradicting the assumed correctness of $\mathcal{A}$. ◀

Summarizing: by definition, $\mathcal{FSTA}^{A_M} \geq \mathcal{FSTA}^A$; by Lemma 8, it follows that TF is solvable in $\mathcal{FSTA}^{A_M}$; and, by Lemma 22, it follows that TF is not solvable in $\mathcal{FSTA}^A$. In other words:

▶ **Theorem 23.** $\mathcal{FSTA}^{A_M} > \mathcal{FSTA}^A$

▶ **Theorem 24.**
1. $\mathcal{FSTA}^{A_{LC}} \equiv \mathcal{FSTA}^A$
2. $\mathcal{FSTA}^S > \mathcal{FSTA}^{A_M}$
3. $\mathcal{FSTA}^S > \mathcal{FSTA}^{A_{LC}}$
4. $\mathcal{FSTA}^{A_M} > \mathcal{FSTA}^{A_{LC}}$

Proof. **1.** holds because, by definition, $\mathcal{FSTA}$ robots cannot distinguish between A_{LC} and A. **2.** follows from follows from Lemmas 19 and 20. **3.** follows from **1.** and Theorem 21. **4.** follows from **1.** and Theorem 23. ◀

6 Relationship Between Models Under Asynchronous Schedulers

In the previous sections, we have characterized the asynchronous landscape within each robot model. In this section, we determine the computational relationship between the different models under the asynchronous schedulers A_{LC}, A_M, A_{CM} and ASYNCH.

We do so by first determining the relationship between $\mathcal{FCOM}$ and the other models under the asynchronous schedulers; we then complete the characterization of the landscape by establishing the still remaining relationships, those between $\mathcal{FSTA}$ and $\mathcal{OBLOT}$.

6.1 Relative power of $\mathcal{FCOM}$

In this section, we determine the relationship between $\mathcal{FCOM}$ and the other models under the asynchronous schedulers A_{LC}, A_M, A_{CM} and ASYNCH.

We first show that $\mathcal{FCOM}^{A_{LC}}$ and $\mathcal{FSTA}^{A_M}$ are orthogonal. To prove this result we use the existence of a problem, `Cyclic Circles` (CYC), shown in [6] to be solvable in $\mathcal{FCOM}^A$ but not in $\mathcal{FSTA}^S$:

▶ **Lemma 25** ([6]).
1. *$CYC \notin \mathcal{FSTA}^S$*
2. *$CYC \in \mathcal{FCOM}^A$, even under non-rigid-movement.*

We then consider the problem `Get Closer but Not too Close on Line` (GCNCL) defined as follows.

▶ **Definition 26** (Get Closer but Not too Close on Line (GCNCL)). *Let a and b be two robots on distinct locations $a(0), b(0)$ where $r(t)$ denotes the position of $r \in \{a, b\}$ at time $t \geq 0$. This problem requires the two robots to get closer, without ever increasing their distance on the line connecting them, and eventually stop at distance at least $|a(0) - b(0)|/2$ from each other.*

In other words, an algorithm solves GCNCL iff it satisfies the following predicate:

$$GCNCL \equiv \left(\forall t \geq 0 : a(t), b(t) \in \overline{a(0)b(0)}\right) \wedge (\forall t, t' : 0 \leq t \leq t' \rightarrow d_t \geq d_{t'}) \\ \wedge \left(\exists t : \frac{d_0}{2} \leq d_t < d_0 \wedge (\forall t' \geq t : a(t) = a(t') \wedge b(t) = b(t'))\right),$$

where d_t is the distance between the two robots at time t, i.e., $d_t = |a(t) - b(t)|$.

▶ **Lemma 27.**
1. *$GCNCL \notin \mathcal{FCOM}^S$.*
2. *$GCNCL \in \mathcal{FSTA}^A$.*

Proof. **1.** The impossibility of $\mathcal{FCOM}^S$ can be obtained as follows. Since we consider $\mathcal{FCOM}$, a robot computes its destination depending on the color of its opponent, not on its own color. We say that a color c is *attractive* if a robot decides to move (i.e., not stay) when the color of the opponent is c. The adversary can prevent the robots from solving GCNCL in the following way. Initially, both robots have the same color. If that color is not attractive, the adversary keeps on simultaneously activating both robots until the color of the robots becomes attractive. During this period, no robot moves by the definition of attractive colors. Note that an attractive color must appear eventually to solve GCNCL. From then on, the adversary keeps on activating only one robot, say a, while never activating b. During this period, b never changes its color, so the color of b is always attractive. Because of variable disorientation, the adversary can guarantee that there is a fixed positive constant $c \leq 1$ such that when a is activated at time t, the resulting distance between a and b (after a moves) is $c \cdot d_t = c \cdot |a(t) - b(t)|$. (The robots immediately violate the specification of GCNCL if $c > 1$ or $c = 0$.) However, this implies that the distance between a and b converges to zero as a moves repeatedly, violating the specification of GCNCL.

2. The problem is easily solvable with $\mathcal{FSTA}$ robot in ASYNCH. Let the robots have color A initially. The first time a robot is activated, it moves closer by distance $d/4$ to the other and changes its color to B, where d is the observed distance. Whenever a robot is activated, if its color is B, it does not move. Clearly, both robots eventually stop and their final distance is at least $d_0/2$. ◀

The orthogonality of $\mathcal{FCOM}^{A_{LC}}$ and $\mathcal{FSTA}^{A_M}$ (or $\mathcal{FSTA}^{A}$) then follows from Lemmas 25 and 27.

▶ **Theorem 28.**
1. $\mathcal{FCOM}^{A_{LC}} \perp \mathcal{FSTA}^{A_M}$
2. $\mathcal{FCOM}^{A_{LC}} \perp \mathcal{FSTA}^{A}$
3. $\mathcal{FCOM}^{A_{LC}} > \mathcal{OBLOT}^{S}$

Proof. **1.**–**2.** By Lemmas 25 and 27. **3.** is proved by the fact that RDV can be solved by $\mathcal{FCOM}^{S}$ but not by $\mathcal{OBLOT}^{S}$, and by the equivalence of $\mathcal{FCOM}^{S}$ and $\mathcal{FCOM}^{A_{LC}}$. ◀

The following theorem shows the relative power of $\mathcal{FCOM}^{A}$.

▶ **Theorem 29.**
1. $\mathcal{FCOM}^{A}(\equiv \mathcal{FCOM}^{A_M}) \perp \mathcal{FSTA}^{A_M}$
2. $\mathcal{FCOM}^{A} \perp \mathcal{FSTA}^{A}$
3. $\mathcal{FCOM}^{A} \perp \mathcal{OBLOT}^{S}$
4. $\mathcal{FCOM}^{A} > \mathcal{OBLOT}^{A_M}$

Proof. **1.** (resp. **2.**) follows from Theorem 28 **1.** (resp. **2.**) and noting that CYC can be solved in $\mathcal{FCOM}^{A}$. **3.** is proved by Lemmas 4 and 13(MLCv can be solved in $\mathcal{OBLOT}^{S}$ but cannot be solved in $\mathcal{FCOM}^{A_M}$) and the fact that CYC cannot be solved in $\mathcal{FSTA}^{S}$ (and so $\mathcal{OBLOT}^{S}$). **4.** is proved by the equivalence of $\mathcal{FCOM}^{A_M}$ and $\mathcal{FCOM}^{A}$ and using the result of RDV. ◀

The relationship between $\mathcal{FCOM}$ and the other models under the asynchronous schedulers has been determined in the previous section (Theorems 28 and 29). To complete the characterization of the relationship between the computational power of the models under the asynchronous schedulers, we need to determine the relationship between $\mathcal{FSTA}$ and $\mathcal{OBLOT}$.

▶ **Theorem 30.**
1. $\mathcal{FSTA}^{A_M} \perp \mathcal{OBLOT}^{S}$
2. $\mathcal{FSTA}^{A_M} > \mathcal{OBLOT}^{A_M} > \mathcal{OBLOT}^{A}$
3. $\mathcal{FSTA}^{A} \perp \mathcal{OBLOT}^{A_M}$
4. $\mathcal{FSTA}^{A} \perp \mathcal{OBLOT}^{S}$
5. $\mathcal{FSTA}^{A} > \mathcal{OBLOT}^{A}$

Proof. Note that RDV can be solved in $\mathcal{FSTA}^{A}$ (and so $\mathcal{FSTA}^{A_M}$) but cannot be solved in $\mathcal{OBLOT}^{S}$ (and so $\mathcal{OBLOT}^{A_M}$ and $\mathcal{OBLOT}^{A}$). **1.** is proved by the results of RDV, and MLCv, which can be solved in $\mathcal{OBLOT}^{S}$ but cannot be solved in $\mathcal{FSTA}^{A_M}$ (Lemmas 4 and 20). **2.** is proved with the result of RDV and Theorem 10. **3.** (resp. **4.**) are proved with the result of RDV and TF (Lemmas 8, 22 and the equivalence of $\mathcal{FSTA}^{A_{LC}}$ and $\mathcal{FSTA}^{A}$) (resp. MLCv (Lemmas 4, 20 and Theorem 23)). **5.** is proved by the result of RDV. ◀

7 Concluding Remarks

In this paper, we investigated the computational relationship between the power of the four models $\mathcal{OBLOT}$, $\mathcal{FSTA}$, $\mathcal{FCOM}$ and $\mathcal{LUMI}$, under a range of asynchronous schedulers, from Ssynch to Asynch, and provided a complete characterization of such relationships. In this process, we have established a variety of results on the computational powers of the robots in presence or absence of (limited) internal capabilities of memory persistence and/or communication. These results include the proof of computational separation between Ssynch and Asynch in absence of either capability, closing several important open questions.

This investigation has also provided valuable insights into the elusive nature of the relationship between asynchrony and the level of atomicity of the *Look*, *Comp*, and *Move* operations performed in an *LCM* cycle. In fact, in this paper, the study of the asynchronous landscapes has focused on precisely the set of asynchronous schedulers defined by the different possible atomic combinations of those operations as well as the *Move* operation: starting from *LCM*-**atomic**-ASYNCH, which corresponds to SSYNCH, ending with ASYNCH, and including *LC*-**atomic**-ASYNCH, *CM*-**atomic**-ASYNCH, and *M*-**atomic**-ASYNCH.

These results open several new research directions. In particular, an important direction is the examination of other classes of asynchronous schedulers, to further understand the nature of asynchrony for robots operating in *LCM* cycles, identify the crucial factors that render asynchrony difficult for the robots, and possibly discover new methods to overcome it.

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.
3. S. Bhagat and K. Mukhopadhyaya. Optimum algorithm for mutual visibility among asynchronous robots with lights. In *Proc. 19th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 341–355, 2017. doi:10.1007/978-3-319-69084-1_24.
4. 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 337–346, 2013. doi:10.1109/ICDCS.2013.27.
5. K. Buchin, P. Flocchini, I. Kostitsyna, T. Peters, N. Santoro, and K. Wada. Autonomous mobile robots: Refining the computational landscape. In *APDCM 2021*, pages 576–585, 2021. doi:10.1109/IPDPSW52791.2021.00091.
6. K. Buchin, P. Flocchini, I. Kostitsyna, T. Peters, N. Santoro, and K. Wada. On the computational power of energy-constrained mobile robots: Algorithms and cross-model analysis. In *Proc. 29th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 42–61, 2022. doi:10.1007/978-3-031-09993-9_3.
7. D. Canepa and M. Potop-Butucaru. Stabilizing flocking via leader election in robot networks. In *Proc. 10th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 52–66, 2007. doi:10.1007/978-3-540-76627-8_7.
8. S. Cicerone, Di Stefano, and A. Navarra. Gathering of robots on meeting-points. *Distributed Computing*, 31(1):1–50, 2018. doi:10.1007/S00446-017-0293-3.
9. 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.
10. R. Cohen and D. Peleg. Convergence properties of the gravitational algorithms in asynchronous robot systems. *SIAM J. on Computing*, 34(6):1516–1528, 2005. doi:10.1137/S009753970444647.
11. 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.
12. G.A. Di Luna, P. Flocchini, S.G. Chaudhuri, F. Poloni, N. Santoro, and G. Viglietta. Mutual visibility by luminous robots without collisions. *Information and Computation*, 254(3):392–418, 2017. doi:10.1016/J.IC.2016.09.005.
13. S. Dolev, S. Kamei, Y. Katayama, F. Ooshita, and K. Wada. Brief announcement: Neighborhood mutual remainder and its self-stabilizing implementation of look-compute-move robots. In *33rd International Symposium on Distributed Computing*, pages 43:1–43:3, 2019. doi:10.4230/LIPICS.DISC.2019.43.

14 P. Flocchini, G. Prencipe, and N. Santoro (Eds). *Distributed Computing by Mobile Entities*. Springer, 2019. doi:10.1007/978-3-030-11072-7.

15 P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Morgan & Claypool, 2012. doi:10.2200/S00440ED1V01Y201208DCT010.

16 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Hard tasks for weak robots: the role of common knowledge in pattern formation by autonomous mobile robots. In *10th Int. Symp. on Algorithms and Computation (ISAAC)*, pages 93–102, 1999. doi:10.1007/3-540-46632-0_10.

17 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.

18 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Arbitrary pattern formation by asynchronous oblivious robots. *Theoretical Computer Science*, 407:412–447, 2008. doi:10.1016/J.TCS.2008.07.026.

19 P. Flocchini, N. Santoro, Y. Sudo, and K. Wada. On asynchrony, memory, and communication: Separations and landscapes. CoRR abs/2311.03328, arXiv, 2023. arXiv:2311.03328.

20 P. Flocchini, N. Santoro, G. Viglietta, and M. Yamashita. Rendezvous with constant memory. *Theoretical Computer Science*, 621:57–72, 2016. doi:10.1016/J.TCS.2016.01.025.

21 P. Flocchini, N. Santoro, and K. Wada. On memory, communication, and synchronous schedulers when moving and computing. In *Proc. 23rd Int. Conference on Principles of Distributed Systems (OPODIS)*, pages 25:1–25:17, 2019. doi:10.4230/LIPICS.OPODIS.2019.25.

22 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.

23 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.

24 A. Hériban, X. Défago, and S. Tixeuil. Optimally gathering two robots. In *Proc. 19th Int. Conference on Distributed Computing and Networking (ICDCN)*, pages 1–10, 2018. doi:10.1145/3154273.3154323.

25 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.

26 D. Kirkpatrick, I. Kostitsyna, A. Navarra, G. Prencipe, and N. Santoro. Separating bounded and unbounded asynchrony for autonomous robots: Point convergence with limited visibility. In *40th Symposium on Principles of Distributed Computing (PODC)*. ACM, 2021. doi:10.1145/3465084.3467910.

27 T. Okumura, K. Wada, and X. Défago. Optimal rendezvous $\mathcal{L}$-algorithms for asynchronous mobile robots with external-lights. In *Proc. 22nd Int. Conference on Principles of Distributed Systems (OPODIS)*, pages 24:1–24:16, 2018. doi:10.4230/LIPICS.OPODIS.2018.24.

28 T. Okumura, K. Wada, and Y. Katayama. Brief announcement: Optimal asynchronous rendezvous for mobile robots with lights. In *Proc. 19th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 484–488, 2017. doi:10.1007/978-3-319-69084-1_36.

29 G. Sharma, R. Alsaedi, C. Bush, and S. Mukhopadyay. The complete visibility problem for fat robots with lights. In *Proc. 19th Int. Conference on Distributed Computing and Networking (ICDCN)*, pages 21:1–21:4, 2018. doi:10.1145/3154273.3154319.

30 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. doi:10.1007/978-3-642-05118-0_47.

31 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.

32 S. Terai, K. Wada, and Y. Katayama. Gathering problems for autonomous mobile robots with lights. *Theoretical Computer Science*, 941(4):241–261, 2023. doi:10.1016/J.TCS.2022.11.018.

33 G. Viglietta. Rendezvous of two robots with visible bits. In *10th Int. Symp. on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS)*, pages 291–306, 2013. doi:10.1007/978-3-642-45346-5_21.

34 M. Yamashita and I. Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theoretical Computer Science*, 411(26–28):2433–2453, 2010. doi:10.1016/J.TCS.2010.01.037.

35 Y. Yamauchi, T. Uehara, S. Kijima, and M. Yamashita. Plane formation by synchronous mobile robots in the three-dimensional euclidean space. *J. ACM*, 64:3(16):16:1–16:43, 2017. doi:10.1145/3060272.

On the Round Complexity of Asynchronous Crusader Agreement

Ittai Abraham
Intel Labs, Petah Tikva, Israel

Naama Ben-David
Technion, Haifa, Israel

Gilad Stern
The Hebrew University of Jerusalem, Israel

Sravya Yandamuri[1]
Duke University, Durham, NC, USA

Abstract

We present new lower and upper bounds on the number of communication rounds required for *asynchronous* Crusader Agreement (CA) and Binding Crusader Agreement (BCA), two primitives that are used for solving binary consensus. We show results for the information theoretic and authenticated settings. In doing so, we present a generic model for proving round complexity lower bounds in the asynchronous setting. In some settings, our attempts to prove lower bounds on round complexity fail. Instead, we show new, tight, rather surprising round complexity upper bounds for Byzantine fault tolerant BCA with and without a PKI setup.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases lower bounds, asynchronous protocols, round complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.29

Related Version *Full Version*: `https://eprint.iacr.org/2023/1586`

Funding *Gilad Stern*: This work was supported by the HUJI Federmann Cyber Security Research Center in conjunction with the Israel National Cyber Directorate (INCD) in the Prime Minister's Office.

1 Introduction

Agreement problems are at the core of many distributed systems, finding applications in replicated and reliable systems, transactional systems, cryptocurrencies, and more. It is therefore not surprising that they have gained a lot of attention in the research community, with tens of papers written about agreement problems each year. A key metric of the performance of many distributed tasks, agreement problems included, is their *round complexity*, or, intuitively, the number of sequential network round trips required to solve the task. In practice, round complexity often translates directly to latency, since communication over distributed networks is slow and forms a major bottleneck in many systems [2, 3, 11, 19, 21, 26, 27, 28, 29].

Arguably the most important and well-known agreement problem, called *consensus*, requires all non-faulty parties to unanimously agree on the same valid input value. Unfortunately, a seminal result of Fischer, Lynch and Paterson shows that no consensus algorithm can guarantee termination in an asynchronous failure-prone system [17]. Interestingly, however, weaker agreement problem variants *can* be solved in such systems, and can be sufficient for many applications.

[1] Lead author.

In one such problem, known as Crusader Agreement, all parties receive an input, and non-faulty parties must output either one of the input values or a special value $\perp$. All non-faulty parties outputting a non-$\perp$ value must agree, and are only allowed to output $\perp$ if there were at least two unique input values among the non-faulty parties [12]. This weakening of consensus can be quite powerful; intuitively, if a non-$\perp$ decision represents an action, it ensures that no conflicting actions will be taken by non-faulty parties. Furthermore, CA and its variants have been used as subroutines to solve consensus in randomized protocols [1, 6, 7, 9, 25].

Our contributions

In this paper, we focus on the *Crusader Agreement (CA)* problem, and present an in-depth study of the achievable round-complexity of the problem and its variants. In particular, we consider classic CA, as well as two important variants: *Binding* Crusader Agreement (BCA) and *Graded* (Binding) Crusader Agreement (G(B)CA). In BCA, crusader agreement must be solved, but with the additional requirement that at the time at which the first non-faulty process decides its output, there exists a non-$\perp$ value v such that no non-faulty party can output a different non-$\perp$ value in any continuation of the execution. Intuitively, the adversary is *bound* to one non-$\perp$ output value and cannot adaptively affect outputs based on future knowledge. This property has recently been shown to be crucial for solving randomized consensus an an asynchronous setting [1]. In GBCA, in addition to binding, *confidence levels* or *grades* are introduced, so that parties outputting a non-$\perp$ value do so with a *grade 1* or *grade 2* label, with the guarantee that if any non-faulty party outputs v with grade 2, no non-faulty party outputs $\perp$. This variant of CA is also useful in solving randomized consensus [1]. For all of these problems, we present lower and upper bounds on their round complexity in the asynchronous model, considering both crash and Byzantine failures. We consider networks with n parties and f faulty parties.

The lower bounds for crash-resilient protocols specifically deal with protocols in which the adversary can adaptively choose the inputs of some of the parties when it schedules their first actions. While this notion of adaptive inputs might seem unnatural, when using binding crusader agreement protocols to construct consensus protocols, it is advantageous to use protocols that are also secure when the adversary is able to choose inputs adaptively, both in terms of efficiency and simplicity. For further discussion on this topic, we refer the reader to the full version of the paper.

We first show that binding crusader agreement (BCA) requires 2 rounds if f parties can crash and $2f + 1 \leq n \leq 3f$ in the adaptive input setting.

▶ **Theorem 1.** *It is impossible to solve crash fault tolerant BCA in* 1 *round when* $2f + 1 \leq n \leq 3f$*, and the adversary can adaptively choose the inputs of the parties.*

We next turn to more complex lower bounds showing tasks where at least 3 rounds are required. First, we show that at least 3 rounds are required for crash-fault resilient graded binding crusader agreement (GBCA) if $2f + 1 \leq n \leq 3f$ in the adaptive input setting.

▶ **Theorem 2.** *It is impossible to solve crash fault tolerant GBCA in* 2 *rounds when* $2f + 1 \leq n \leq 3f$*, and the adversary can adaptively choose the inputs of the parties.*

Protocols solving crash-fault tolerant BCA in 2 rounds and crash-fault tolerant GBCA in 3 rounds have been constructed in [1], showing that these lower bounds are tight.

Next, we show that at least 3 rounds are required for solving Byzantine-fault tolerant crusader agreement (CA) if there is no PKI setup and $3f + 1 \leq n \leq 4f$.

▶ **Theorem 3.** *It is impossible to solve Byzantine fault tolerant CA in 2 rounds when* $3f + 1 \leq n \leq 4f$ *without PKI.*

We also show that this lower bound is tight in the full version of this paper. Lastly, we show that the same bound holds for Byzantine-fault tolerant binding crusader agreement (BCA) if there is a PKI setup and $f \geq 2$, $3f + 1 \leq n \leq 4f$.

▶ **Theorem 4.** *It is impossible to solve Byzantine fault tolerant BCA in 2 rounds with PKI when* $3f + 1 \leq n \leq 4f$ *and* $f \geq 2$.

The lower bounds are first proven for one (or two) failures and then generalized to an arbitrary number of failures. Somewhat surprisingly, for our lower bounds that start with $f = 2$, the generalization to arbitrary $f > 2$ requires a non-trivial argument, requiring both a stronger lower bound for the $f = 2$ case and a more intricate method of generalization (see Appendix B).

Our Contributions: Upper Bounds

While thinking through the aforementioned lower bounds, some bounds seemed elusive and quite hard to achieve. This led us to the discovery of some surprising upper bounds. For example, the final lower bound described in the previous section looks suspiciously different from the other bounds: it only holds when $f \geq 2$. It turns out that the reason a more general lower bound couldn't be constructed is that there exists a protocol solving Byzantine-fault tolerant binding crusader agreement in 2 rounds if there is a PKI setup and $n = 4, f = 1$! Following this discovery, we constructed two more protocols that work for a small number of parties but don't seem to obviously generalize to any n and f. More precisely, we construct protocols solving Byzantine-fault tolerant binding crusader agreement in 3 rounds without a PKI setup for $n = 4, f = 1$ and for $n = 7, f = 2$. The resulting protocol is also a 3-round Byzantine-fault tolerant crusader agreement protocol for any n, providing a matching upper bound to one of above lower bounds.

A key insight to constructing these protocols is to design them to be as *patient* and *conservative* as possible. By *conservative*, we mean that parties output a non-$\perp$ value only if they have to. More concretely, they output the value v only if they see that their view could have been generated in a run in which all nonfaulty parties had the input v. In this case, parties must output v; otherwise, they may violate the validity of the protocol in some run. In all other cases, parties output $\perp$. By *patient*, we mean that parties wait and output a value only when they absolutely have to. More precisely, we aim to have parties output a value only when their view could have been generated in a run of the protocol in which they may not receive any more messages. Clearly, if they do not output a value at that point, there is a run in which they never output a value. This allows us to gather as much information as possible before parties output some value.

A somewhat surprising realization is that many protocols aren't as patient as they are allowed to be. For example, many protocols simply wait to hear $n - f$ messages in a given round before proceeding to the next. On the other hand, patient protocols could wait for even more information. For example, in the second round of the protocol, parties could wait to hear both round 1 and round 2 messages from the *same* $n - f$ parties, and for each others' reports to be consistent. From our upper bounds it seems like these conditions can be quite intricate and potentially very expensive to compute for large values of n. As such, we don't suggest these protocols as realistic upper bounds, but rather almost as an impossibility result, showing that a lower bound cannot be constructed for these cases. In further work, we hope to either show that these upper bounds are general, or that a lower bound can be constructed for some $f \geq 3$.

Related Work

It is well known that there are many impossibility results and lower bounds on distributed protocols [23]. Early results in the field show lower bounds on the round complexity in synchronous networks. For example, Fischer and Lynch show that $f+1$ rounds are needed to reach Byzantine consensus in [16]. This lower bound was later generalized to authenticated protocols in [10] and [14]. In addition, similar lower bounds have been shown for synchronous crash-resilient consensus [4, 15]. Bounds are also known on early-stopping consensus, showing that if the number of actually faulty parties is smaller than the corruption threshold, the number of needed rounds is at least 2 more than the number of corrupted parties [13].

On the other hand, fewer lower bounds are known on the round complexity of asynchronous protocols. The FLP result [17] shows that no deterministic consensus algorithm exists in an asynchronous system, even in the face of a single crash failure. More precisely, the proof shows that any consensus protocol in this setting has an infinite execution, essentially showing that the round complexity of such protocols is infinite. Similarly, the CAP theorem states that no distributed database can have consistency, availability and resilience to network partitioning [18, 24].

Concurrent work by Attiya and Welch deals with a new primitive called connected consensus [5]. This primitive generalizes both crusader agreement and graded crusader agreement. In this work, Attiya and Welch construct several protocols solving this task, and provide lower bounds on the efficiency of such protocols. For example, they construct unauthenticated protocols solving binding crusader agreement and graded binding crusader agreement in 1 and 2 rounds respectively, as well as several other protocols. Note that their protocols do not consider adaptive inputs, meaning they can avoid the lower bounds of Theorems 1 and 2. In addition, their work includes lower bounds showing that GBCA requires 2 rounds in the case of crash failures with $n \leq 4f$ and in the case of Byzantine failures with $n \leq 9f$.

2 Model & Definitions

2.1 Model

Network

This work deals with a network of n parties connected via point-to-point communication channels. The network is asynchronous, meaning that there is no bound on message delay, but every message is eventually delivered in finite time. We assume that the point-to-point channels deliver messages in a FIFO order. The means that if a party sends a message m and then a message m' to the same party, the messages are delivered in that order. This can be enforced by simply adding a counter to each message, signifying when it was sent.

We model message delivery as being controlled by an adversary that can choose any delivery schedule as long as all messages are eventually delivered. We consider two types of faults in this work: crash and Byzantine faults. In networks with crash faults the adversary may cause up to f parties to crash, meaning that those parties do not take any further actions (including receiving or sending messages). On the other hand, in networks with Byzantine faults the adversary can control up to f parties and cause them to deviate arbitrarily from the protocol.

Finally, when we say that a network has a PKI setup, we mean that each party has a well-known public key and a private key that allow it to sign messages. Every party can use the public key to check that a message was indeed sent by a given party. In addition, parties can forward received messages with their signatures, proving that the message was indeed sent by the signing party.

Asynchronous Rounds

In the synchronous setting, rounds are very clearly defined using the bound Δ on message delivery. Defining the notion of round complexity for asynchronous protocols is less straightforward [8, 20, 22], and we follow [22]. We use the idea of "causal chains" in our definition of asynchronous round complexity. Intuitively, we can think of chains of messages, with each message being sent as a result of receiving previous messages. When a message is sent, it lengthens its chain by 1, and it is considered a round k message if its chain is of length k. When mapping this behaviour to synchronous systems, all of the messages that are sent without receiving any message will be sent in round 1. Round 2 messages will be sent after receiving round 1 messages, etc.

More precisely, if a message is sent in the beginning of the protocol without receiving any other message, we consider it to be a round 1 message. If a message is sent by a nonfaulty party as a result of receiving all messages in a set M, we consider it a round $k+1$ message, where k is the maximal round number for nonfaulty messages in M (or $k=0$ if there is no such message). We say that a party is in round k if it sent or received at least one round k message, and did not send or receive any higher-round message.

Using this notion of round complexity, we can define a k-round protocol:

▶ **Definition 5** (k-Round Protocol). *A protocol is a k-round protocol if all honest parties decide a value after at most k rounds.*

Note that it is possible that protocols never terminate or do not have a bound k on the number of rounds. If this happens, these protocols can be defined as having infinite round complexity, but we deal only with finite round complexity protocols in this work.

Adaptive Inputs

We say that an adversary can choose inputs adaptively if parties only have their inputs defined by the adversary at the moment they start participating in the protocol. When dealing with binding protocols, to be defined below, this means that the binding values can only depend on the state of the nonfaulty parties that started participating in the protocol at that time, and cannot depend on the inputs of parties that haven't started participating in the protocol.

2.2 Definitions

We start by defining the different tasks for which we have constructed lower and upper bounds. In this work we only consider protocols in which parties decide on values but continue sending messages even after their decision. This is a very common technique in the design of asynchronous protocols, allowing parties to help each other even after they have all the information needed to complete the protocols.

▶ **Definition 6** (Crusader Agreement (CA)). *In a Crusader Agreement protocol, each party has either 0 or 1 as an input, and parties decide either 0, 1 or $\perp$. A Crusader Agreement protocol has the following properties:*

(Agreement) *If two nonfaulty parties decide values x and y, then either $x=y$ or one of the values is $\perp$.*

(Validity) *If all nonfaulty parties have the same input, then this is the only possible decision for nonfaulty parties.*

(Termination) *All nonfaulty parties eventually decide.*

To be able to implement CA with an optimal tolerance to crash faults, we must weaken its validity property to the following:

- **(Weak Validity)** If all parties have the same input v, then all nonfaulty parties decide v.

▶ **Definition 7** (Graded Crusader Agreement (GCA)). *In a Graded Crusader Agreement protocol, each party has either* 0 *or* 1 *as an input, and parties decide on pairs* (v, g) *such that* $v \in \{0, 1, \perp\}, g \in \{0, 1, 2\}$ *and* $v = \perp$ *if and only if* $g = 0$*. A Graded Crusader Agreement protocol has the following properties:*

(Graded Agreement) *If two nonfaulty parties decide on the pairs* $(v, g), (v', g')$*, then* $|g - g'| \leq 1$ *and if* $v \neq v'$*, either* $v = \perp$ *or* $v' = \perp$*.*

(Validity) *If all nonfaulty parties have the same input* v*, then all nonfaulty parties decide* $(v, 2)$*.*

(Termination) *All nonfaulty parties eventually decide.*

We define crash fault tolerant CA by weakening the validity property as with the non-graded version. We are also interested in the binding versions of both of these protocols. These protocols add an additional requirement that once the first nonfaulty party completes the protocol, the decision values are "bound". In a BCA protocol this means that even if the first party decides $\perp$, at that time we know which is the only possible non-$\perp$ decision value.

▶ **Definition 8** (Binding Crusader Agreement (BCA)). *A Binding Crusader Agreement protocol has all of the properties of a Crusader Agreement protocol as well as the following property:*

(Binding) *At the time at which the first nonfaulty party to decide decides on a value, there exists a value* $b \in \{0, 1\}$ *such that no nonfaulty party decides* $1 - b$ *in any extension of this execution.*

Note that the binding property is only interesting in the case that the nonfaulty party referred to in the definition decided $\perp$. Otherwise, it trivially follows from agreement. Like in the binding definition of crusader agreement, once the first nonfaulty party decides on a value in a graded binding crusader agreement protocol, there is only one non-$\perp$ value that can be output from the protocol (with some grade).

▶ **Definition 9** (Graded Binding Crusader Agreement (GBCA)). *A Graded Binding Crusader Agreement protocol has all of the properties of a Graded Crusader Agreement protocol as well as the following property:*

(Graded Binding) *At the time at which the first nonfaulty party to decide decides on a value, there exists a value* $b \in \{0, 1\}$ *such that no nonfaulty party decides either* $(1 - b, 2)$ *or* $(1 - b, 1)$ *in any extension of the protocol.*

We define crash fault tolerant BCA and GBCA by weakening the validity property as with the non-graded version.

3 Lower Bounds

General Proof Approach

Each of the presented lower bounds is proven in two steps. We start by proving a lower bound for a small number of parties, setting f to be 1 or 2. We then generalize these proofs in Appendix B. We show that if a protocol exists for some larger values of n and f, then such a protocol exists for the n and f for which we proved the original lower bound with the same round complexity. This is done by assuming that more general protocols exist, and showing that parties can simulate these protocols in the original settings (with a smaller number of parties).

For the proof of each lower bound, we construct a series of worlds. The worlds are constructed strategically to show that a party must take a certain action because their view is indistinguishable from another world where taking a different action would violate some property. In particular, we show indistinguishability with worlds where (1) all (nonfaulty) parties start with the same value, so deciding a different value would result in a violation of validity, and (2) all nonfaulty parties have sent all possible messages, so waiting for additional messages before deciding would result in a violation of termination. We put the descriptor "nonfaulty" in parenthesis where relevant due to the difference in the validity condition for crash and Byzantine fault tolerant protocols. To give the reader a hint as to the purpose of each world in our proofs, we add certain labels to the worlds.

We now describe the labels. In an **x-validity** world, all (nonfaulty) parties have input value x. In a **false x-validity** world, the view of some (nonfaulty) party is the same as in an x-validity world, causing them to decide a non-$\perp$ value (and grade 2, where relevant) even though all (nonfaulty) parties did not have the same input values. In a **maximally patient** world, a party receives all the messages that will be sent to them by nonfaulty parties, and therefore must decide without waiting for additional messages that depend on the actions of faulty parties. For the maximally patient label, we also indicate the party that crashes, meaning another party cannot wait for messages that depend on this party before deciding without violating termination. In a **false maximally patient** world, a nonfaulty party's view is the same as in a maximally patient world, so they decide before receiving all of the messages sent by nonfaulty parties. As previously mentioned, our proofs generally proceed by constructing a chain of worlds, where there are "validity worlds" on opposite ends, and in the middle of the chain some property (binding or agreement) is violated. We indicate when a world is **symmetric** to another previously-described world on the opposite end of the chain. We use the labels **binding violation** and **agreement violation** to indicate worlds in which the properties of binding and agreement are violated, respectively.

In addition to using labels, we separate the description of each world into two bullets. The first bullet indicates the messages sent by the parties and any message delays or specific orderings where needed. The second bullet indicates the view of one or more nonfaulty parties and the actions they take accordingly.

3.1 Results

For our first result, we start with a simple 1 round lower bound for crash fault tolerant BCA with adaptive inputs.

▶ **Theorem 1.** *It is impossible to solve crash fault tolerant BCA in* 1 *round when* $2f + 1 \leq n \leq 3f$*, and the adversary can adaptively choose the inputs of the parties.*

We show a proof for a network of three parties: p_1, p_2, and p_3. Our ultimate goal is to build up to **World 4**, in which binding is violated. In **World 4**, a party decides while p_3 lags behind; after this, the adversary adaptively chooses the input of p_3 and forces p_3 to decide 1 or 0 after a party has already decided. In order to show why p_3 decides 1 or 0 in those executions, we show indistinguishability from **World 1** or **World 2**, where all parties start with input 1 or 0, respectively. In those worlds, p_3 must decide 1 or 0 in order to not violate validity. To show why the first-deciding party decides in **World 4** without waiting for any messages from p_3, we show indistinguishability from **World 3**, in which p_3 crashes without sending any messages. In **World 3**, parties cannot wait for messages that are dependent on p_3 before deciding, as this would result in a violation of termination.

3 party proof.

World 1 ($1-$validity, maximally patient for p_2 crash):

- p_1 and p_3 are nonfaulty. p_2 crashes immediately. All parties have input 1.
- p_1 and p_3 must decide 1 after receiving each other's messages without waiting for any additional messages by validity and termination.

World 2 ($0-$validity, maximally patient for p_1 crash):

- p_2 and p_3 are nonfaulty. p_1 crashes immediately. All parties have input 0.
- p_2 and p_3 must decide 0 after recieving each other's messages without waiting for any additional messages by validity and termination.

World 3 (maximally patient for p_3 crash):

- p_1 and p_2 are nonfaulty. p_3 crashes immediately. p_1 and p_3 start with inputs 1 and 0 respectively.
- p_1 and p_2 must decide after receiving each other's messages without waiting for any additional messages by termination.

World 4 (false maximally patient, false validity, binding violation):

- p_1, p_2, and p_3 are nonfaulty. p_1 starts with input 1 and p_2 starts with input 0; p_3 lags behind, and its input will be adaptively chosen later. p_1 and p_2's messages are delivered to each other, so they decide due to indistinguishability from **World 3**. The adversary now chooses one of the following extensions:
  1. p_3 has input value 1. p_1's messages are delivered to p_3, and p_2's messages are only delivered after p_3 decides.
  2. p_3 has input value 0. p_2's messages are delivered to p_3, and p_1's messages are only delivered after p_3 decides.
- In extension 1, p_3 outputs 1 due to indistinguishability from **World 1**; or in extension 2, p_3 outputs 0 due to indistiguishability from **World 0**. This constitutes a binding violation, as we show that both 1 or 0 are possible values that p_3 decides after another party has already decided. Note that this does not imply a violation of agreement, as it is possible for the party (or parties) deciding before p_3 to decide $\perp$. ◀

We now present our second result in the crash case: a 2 round lower bound for GBCA.

▶ **Theorem 2.** *It is impossible to solve crash fault tolerant GBCA in* 2 *rounds when* $2f+1 \leq n \leq 3f$*, and the adversary can adaptively choose the inputs of the parties.*

We show a proof using a network of three parties: p_1, p_2, and p_3. Our approach is to build up to a world, **World 3**, in which there is a violation of binding. The strategy of the adversary to violate binding is as follows. First, p_1 is forced to output before p_3's input value is chosen. Then, the adversary chooses p_3's input and forces them to decide 1 or 0, thus breaking binding. To show how the adversary has p_3 decide 1 or 0 in **World 3**, we present 2 symmetric sets of 3 worlds. Each set consists of the following three types of worlds:

1. A validity world showing why a party must decide a non-$\perp$ value with grade 2
2. A world where one of the parties crashes
3. A world that is both indistinguishable from the first type of world for some party other than p_3 (meaning that it decides a non-$\perp$ value with grade 2) and indistinguishable from the second type of world for p_3, showing why p_3 decides the non-$\perp$ value that it does (so as not to violate graded agreement) in each extension of **World 3** without waiting for more messages (so as not to violate termination).

For ease of exposition, we include only the worlds described in point 3 above (**World 1** and **World 2**) in the main proof of this theorem. We separate the indistinguishability arguments and the corresponding worlds into two lemmas: Lemma 10 and 12. Apart from the 2 sets of

3 symmetric worlds described above, and **World 3** in which binding is broken, we construct an additional world to show why p_1 decides in **World 3** while p_3 lags behind. This world and the corresponding indistinguishability argument are proven separately in Lemma 13. We provide the proof of the first lemma after the proof of Theorem 2 and refer the reader to Appendix A for similar proofs of the next two lemmas.

3 party proof. In the description of the following worlds, we only describe the runs until a specific point, and have some arbitrary message scheduling following that.

World 1 (false 1-validity, false maximally patient):

- p_1, p_2, and p_3 are nonfaulty. p_1 and p_3 have input 1, while p_2 has input 0. Initially, p_1's round 1 messages are delivered to p_2 and p_3, and then p_3's round 1 messages are delivered to p_1 and p_2. Following that, any round 2 messages that p_1 sends are delivered to p_2, and any of p_3's round 2 messages are delivered to p_1 and p_2. From this point on, p_2 and p_3's messages are delivered to each other without delay.
- By Lemma 10, p_3 decides without waiting for additional messages, and its output is of the form $(1, g)$ such that $g \in \{1, 2\}$.

World 2 (false 0-validity, false maximally patient, symmetric to World 1):

- p_1, p_2 and p_3 are nonfaulty. p_1 has input 1, and p_2 and p_3 have input 0. Initially, p_2's round 1 messages are delivered to p_1 and p_3, and then p_3's round 1 messages are delivered to p_1 and p_2. Following that, any round 2 messages that p_2 sends are delivered to p_1, and any of p_3's round 2 are delivered to p_1 and p_2. From this point on, p_1 and p_3's messages are delivered to each other without delay.
- By Lemma 12, p_3 must decide $(0, g)$ for $g \in \{1, 2\}$.

World 3 (binding violation, false maximally patient):

- p_1, p_2 and p_3 are nonfaulty. p_1 has input 1, p_2 has input 0, and p_3's input will be adaptively chosen by the adversary based on the value it wants p_3 to output after the first party to output does so. At the start of the execution, p_1 and p_2's round 1 messages are delivered to each other, and then any resulting round 2 messages are delivered to each other. By Lemma 13, p_1 outputs without waiting for any messages that depend on p_3 at this time. We will now show two extensions of this run, one in which p_3 outputs $(1, g)$ for some $g \in \{1, 2\}$, and one in which it outputs $(0, g)$ for some $g \in \{1, 2\}$, showing that the protocol is not binding.
  1. The adversary adaptively chooses input 1 for p_3. Following that, p_3 receives p_1's round 1 messages, and then continues communicating freely with p_2 without any delays. At this point in time, p_3's view consists of round 1 messages from p_1 and p_2 and any round 2 messages from p_2 sent as a result as receiving p_1's round 1 messages and then p_3's round 1 messages. This view is identical to the one it has in **World 1**, so p_3 decides $(1, g)$ for some $g \in \{1, 2\}$.
  2. The adversary adaptively chooses input 0 for p_3. Following that, p_3 receives p_2's round 1 messages, and then continues communicating freely with p_1 without any delays.
     At this point in time, p_3's view consists of round 1 messages from p_1 and p_2 and any round 2 messages from p_1 sent as a result as receiving p_2's round 1 messages and then p_3's round 1 messages. This view is identical to the one it has in **World 2**, so p_3 decides $(0, g)$ for some $g \in \{1, 2\}$. ◀

▶ **Lemma 10.** *In **World 1** from the proof of Theorem 2, p_3 must decide (1, g) for $g \in \{1, 2\}$ without waiting for any round 2 messages from p_1.*

Proof.

World 1.a) (1-validity, maximally patient for p_2 crash):

- p_1 and p_3 are nonfaulty. p_2 crashes without sending any initial messages. All three parties start with input 1. p_1 and p_3 communicate without delay.
- p_1 and p_3 must decide $(1, 2)$ without waiting for any messages from p_2 by validity and termination.

World 1.b) (maximally patient for p_1 crash):

- p_1 and p_3 have input 1, while p_2 has input 0. p_1 is faulty, sends round 1 messages, which are delivered to both p_2 and p_3, and then p_1 crashes. Following that, p_3's round 1 messages are delivered to p_2. Finally, p_2 and p_3's messages are delivered to each other without delay.
- Because p_1 crashed, p_2 and p_3 must decide without waiting for any round 2 messages sent by p_1, by termination.

We now argue why in **World 1** from the proof of Theorem 2, p_3 must decide $(1, g)$ such that $g \in \{1, 2\}$ without waiting for any round 2 messages from p_1. First, we show that p_1 decides (1, 2), in **World 1**. Observe that p_1's view in **World 1** is indistinguishable from its view in **World 1.a** because p_1 and p_3 have input 1 and they start by exchanging both round 1 and round 2 messages. It follows that p_1 decides (1, 2), and thus when p_3 decides some value, it must decide $(1, g)$ such that $g \in \{1, 2\}$ by graded agreement. Next, we argue that p_3 must decide in **World 1** without waiting for any round 2 messages from p_1. Observe that in **World 1**, since p_1's messages (apart from any round 1 messages) are delayed for p_3, p_3's view is indistinguishable from its view in **World 1.b**. As a result, p_3 must not wait for any round 2 messages from p_1 before deciding so as not to violate termination. Note that p_2 cannot send any messages which rely on p_1's round 2 messages, because this is a 2-round protocol, so p_3's view is indeed indistinguishable in both worlds. ◀

For our third result, we show a lower bound for Byzantine fault tolerant CA without PKI. With a Byzantine adversary and no PKI, the faulty parties are able to simulate receiving certain messages from nonfaulty parties.

▶ **Theorem 3.** *It is impossible to solve Byzantine fault tolerant CA in 2 rounds when $3f + 1 \leq n \leq 4f$ without PKI.*

We present a proof for 4 parties: p_1, p_2, p_3 and p_4. In this proof, we build up to **World 5** in which agreement is violated because nonfaulty parties p_1 and p_4 decide 1 and 0, respectively. We start by showing two maximally patient worlds (**World 1** and **World 2**), where one party has omission failures and sends its input value message only to one other party. By termination, the nonfaulty parties must not wait to hear more messages before deciding. We then show two symmetric validity worlds (**World 3** and **World 4**) in which a Byzantine party simulates receiving a message from a non-faulty party that it didn't send. Due to indistinguishability from the maximally patient worlds, honest parties must decide without waiting for additional messages, but they must decide non-$\perp$ values by validity. Finally, in **World 5**, the adversary uses a Byzantine p_3 to have p_1 and p_4 decide different non-$\perp$ values using indistinguishability from the previously defined worlds.

4 party proof. In the following discussion, when we say that parties p_1, p_2 and p_3 have each other's messages delivered, we mean that the party receives its own messages first, and then p_1's messages are delivered first, then p_2's and then p_3's (similarly for p_2, p_3 and p_4).

World 1 (maximally patient for p_4 crash):

- All parties except p_4 are nonfaulty. p_4 crashes immediately without sending any messages. p_1 and p_2 have input 1; p_3 and p_4 have input 0. p_1, p_2 and p_3 have their round 1 messages delivered to each other, and then any round 2 messages that they send as a result are delivered to each other.
- All nonfaulty parties must decide without waiting for any messages dependent on p_4.

World 2 (maximally patient for p_1 omission, symmetric to World 1):

- All parties other than p_1 are nonfaulty; p_1 has omission failures. p_1 and p_2 have input 1, while p_3 and p_4 have input 0. p_1 sends round 1 messages as an honest party would with input 1 only to party p_2, and the messages are delivered first for p_2. Following that, p_2, p_3 and p_4 have their round 1 messages delivered to each other, and then any round 2 messages that they send as a result are delivered to each other.
- All nonfaulty parties must decide without waiting for any more messages from p_1 by termination.

World 3 (0-validity, false maximally patient, simulation):

- All parties except for p_2 are nonfaulty. p_2 is Byzantine. p_1, p_3 and p_4 start with 0. p_2 acts as if it started with input 1 and simulates p_1 starting with input 1. All messages from p_1 are delayed to p_3 and p_4, until they both decide. p_2 acts as if it is a nonfaulty party with input 1 such that the first message it received was a round 1 message from an honest p_1 with input 1. Following that, p_2, p_3 and p_4 have their round 1 messages delivered to each other, and then any round 2 messages that they send as a result are delivered to each other.
- Due to indistinguishability from **World 2**, p_4 decides without waiting for any additional messages. By validity, p_4 decides 0.

World 4 (1-validity, false maximally patient, simulation, symmetric to World 3):

- p_3 is Byzantine, and the remaining parties are nonfaulty. p_1, p_2, and p_4 start with input 1; p_3 acts as if it nonfaulty and has the input 0. All messages from p_4 are delayed to p_1 and p_2. p_1, p_2 and p_3 have their round 1 messages delivered to each other, and then their round 2 messages delivered to each other.
- Due to indistinguishability from **World 1**, p_1 decides before receiving any messages from p_4. By validity, p_1 decides 1.

World 5 (agreement violation, false maximally patient, false validity):

- p_3 is Byzantine, and the remaining parties are nonfaulty. p_1 and p_2 have input 1, while p_3 and p_4 have input 0. p_3 starts by acting as a nonfaulty party would with input 0. Parties p_1, p_2 and p_3's round 1 messages are delivered to each other, and then any round 2 message that they sent as a result of receiving the round 1 messages. Following that, p_3 acts as if it did not receive any round 1 messages from p_1. Now, p_4's round 1 messages are delivered to p_2 and p_3, and their round 1 messages are delivered to p_4. Finally, all round 2 messages sent by p_2 and p_3 are delivered to p_4.
- This world is indistinguishable from **World 4** for p_1 since it exchanged round 1 and round 2 messages with parties p_2 and p_3 with the same inputs without hearing from p_4. In addition, this world is indistinguishable from **World 3** for p_4 because p_1 acts as if it first received round 1 messages from p_1 with input 1, and then p_2, p_3 and p_4 exchange round 1 and round 2 messages without receiving any further messages from p_1. Therefore, p_1 and p_4 decide 1 and 0 respectively, violating the agreement property. ◀

For our second lower bound in the Byzantine case, we prove the impossibility of Byzantine fault tolerant BCA with PKI in 2 rounds when $f \geq 2$. Since there is PKI, the faulty parties can no longer simulate receiving messages from nonfaulty parties. This necessitates a slightly more complex approach than that required for the previous lower bound.

▶ **Theorem 4.** *It is impossible to solve Byzantine fault tolerant BCA in 2 rounds with PKI when* $3f + 1 \leq n \leq 4f$ *and* $f \geq 2$.

In this proof, we build up to a **World 6** where we show a binding violation by having an extension where a nonfaulty p_1 decides 1 and an extension where a nonfaulty p_7 decides 0 after another nonfaulty party p_5 decides. Unlike in the proof of the previous lower bound, we can no longer rely on simulation due to the presence of PKI. If we want a nonfaulty party to decide a non-$\perp$ value $v \in \{0, 1\}$, it can hear that at most $f = 2$ parties started with $1 - v$. This is because, in order to argue that a party must decide a non-$\perp$ value in a given world, we show that this party's view is indistinguishable from its view in another world in which all nonfaulty parties started with that value, enabling us to invoke validity. With PKI, if a party hears that more than f parties started with the value opposite its input value, then it knows that it is not in a validity world. As such, when attempting to understand this proof it is helpful to work backwards, starting from **World 6** to see the views of p_1 and p_7 when they decide 1 and 0, respectively. The maximally patient worlds **World 1**, **World 2**, and **World 5** show why p_1, p_5, and p_7 decide without waiting for additional messages in **World 6**. To show why the views of p_1 and p_7 are indistinguishable from validity worlds, forcing them to decide 1 and 0 respectively, we show **World 3** and **World 4** in which the honest parties all start with the same value.

Proof. As in previous proofs, when we say a party receives messages from a list of parties, they receive the messages in the listed order. For example, if a party receives messages from $p_1, \ldots, p_4$, it receives the messages from p_1 first, then p_2, and so on.

World 1 (maximally patient for p_2 and p_1 crash):

- All parties except p_1 and p_2 are nonfaulty. p_1 and p_2 crash immediately without sending any messages. p_3 and p_4 start with input 1, while p_5, p_6 and p_7 start with input 0.
- All nonfaulty parties must decide without waiting for any messages dependent on p_1 or p_2; otherwise, termination is violated.

World 2 (maximally patient for p_5 crash and p_6 omission):

- All parties except p_5 and p_6 are nonfaulty. p_1, p_2, p_3, and p_4 start with input 1. p_6 and p_7 start with input 0. p_5 crashes immediately without sending any messages. p_6 is omission failure; all messages except for any round 1 messages it sends to p_2 are omitted, and these messages are delivered for p_2 before any messages from any other parties.
- Nonfaulty parties must decide without waiting for any messages dependent on p_5 or any messages dependent on p_6 (other than any round 1 messages it sends to p_2); otherwise, termination is violated.

World 3 (0-validity, false maximally patient):

- p_3 and p_4 are Byzantine and have input 1. The rest of the parties are honest and start with input 0. All messages from p_1 and p_2 are delayed for the other parties. p_3, p_4, p_5, p_6 and p_7 exchange the same messages as in **World 1** and in the same order.
- This world is indistinguishable from **World 1** for p_7. Therefore, it decides without waiting for any additional messages. By validity, p_7 decides 0.

World 4 (1-validity, false maximally patient):

- p_6 and p_7 are Byzantine and start with input 0; the rest of the parties are honest and start with input 1. All messages from p_5 are delayed for the other parties. p_6 doesn't send any messages except for any round 1 messages that it would have sent to p_2 if it was honest, and as in **World 2**, this message is delivered for p_2 before any messages from any other parties. p_1, p_2, p_3, p_4 and p_7 send the same messages in the same order as in **World 2**.
- The world is indistinguishable from **World 2** for p_1, so it decides without waiting for any additional messages. By validity, p_1 decides 1.

World 5 (maximally patient for p_7 and p_1 omission):

- All parties except for p_1 and p_7 are nonfaulty. $p_1, \ldots, p_4$ start with input 1 and $p_5, \ldots, p_7$ start with input 0. All honest parties start by sending their round 1 messages. p_7 crashes immediately after sending its round 1 messages to all of the other parties. p_1 is omission failure, and the only message it sends is its round 1 message to p_2. p_2 receives round 1 messages from p_6 first, then from $p_1, \ldots, p_4$ and p_7, and finally from p_5. p_2 sends round 2 messages as a result of receiving the aforementioned round 1 messages. Parties $p_3, \ldots, p_6$ receive round 1 messages from $p_3, \ldots, p_7$ and send any resulting round 2 messages. They receive any round 1 messages from p_2 following that, and possibly send additional round 2 messages. Finally, p_5 receives all round 2 messages from parties $p_2, \ldots, p_6$.
- Note that parties $p_2, \ldots, p_6$ received all round 1 messages sent by each other, and p_5 received any round 2 message sent as a result from these parties as well. This means that p_5 receives all messages from nonfaulty parties in this world, and thus by termination, p_5 decides without waiting for any additional messages.

World 6 (binding violation, false maximally patient):

- p_3 and p_4 are Byzantine, and the remaining parties are nonfaulty. $p_1, \ldots, p_4$ have the input 1 and $p_5, \ldots, p_7$ have the input 1, like **World 5**. Initially, all messages from other parties are delayed for p_7 and p_1. In addition, messages from p_1 are delayed for $p_3, \ldots, p_6$. The beginning of the run is exactly the same the run in **World 5** for $p_2, \ldots, p_6$, with p_3, p_4 sending the required messages only to parties $p_2, \ldots, p_6$ and not to p_1, p_7. Since p_5's view is identical to one which causes it to decide, it decides some value in this world as well. Next, we show the two executions in which the adversary can get p_1 to decide 1 or p_7 to decide 0, which would mean the protocol isn't binding.
  - **(Extension where p_1 decides 1)** p_1 and p_7 start by receiving round 1 messages from $p_1, \ldots, p_4, p_7$. p_1 then receives any round 2 messages from $p_1, \ldots, p_4, p_7$ except for p_2 final round 2 message sent by p_2 as a result of receiving p_5's round 1 message (which it received last). In the above, p_3 and p_4 are Byzantine, and they only send p_1 the round 2 messages they would have as a result of receiving round 1 messages from $p_1, \ldots, p_4, p_7$. Note that p_1 receives round 1 messages from $p_1, \ldots, p_4, p_7$ and then round 2 messages from $p_1, \ldots, p_4, p_7$ corresponding to p_2 receiving p_6's round 1 messages first, and then all of the parties receiving round 1 messages from each other. p_1's view is identical to the view it would have in **World 4**, so it decides 1.
  - **(Extension where p_7 decides 0)** p_7 sees round 1 messages from $p_3, \ldots, p_6$, and then all round 2 messages that they sent as a result of receiving round 1 messages from $p_3, \ldots, p_7$. Note that they received round 1 message from p_1, p_2 only after receiving those messages. At this point, p_7's view is identical to its view in **World 3**, so it decides 0. ◀

▶ Remark 11. It is possible to define $S = \{p_2, p_3, p_5, p_7\}$ and $T = \{p_1, p_4, p_6\}$. For these sets, $S \cup T = \{p_1, \ldots, p_7\}$, $S \cap T = \emptyset$ and $|S| = 4$, $|T| = 3$. In the proof of Theorem 4, the adversary always corrupts at most one party in S and one party in T. From Theorem 15 we can conclude that no 2-round Byzantine fault tolerant protocol exists even for any $3f + 1 \leq n \leq 4f$ and $f \geq 2$.

4 Upper Bounds

Notation

The notation for a message from a party p_i is i. The initial message from a party is a special case, as it also contains a subscript $v \in \{0, 1\}$ indicating the party's input value. The first message in a valid chain of messages is always an initial message of this form. Chains of messages are separated by the operator $\cdot$. As an example, $\langle i_1 \cdot j \rangle$ is a length two chain where p_j is forwarding the initial message of p_i, where p_i has input value 1. We define the notion of a prefix of a chain recursively. Message chain C' is a prefix of chain C if $C' = C$ or there exists a party p_j such that $\langle C' \cdot j \rangle = \langle C \rangle$. We say that a message chain C *depends on* party p_i if the first message in the chain is of the form i_x such that $x \in \{0, 1\}$ or there exists a prefix of chain C, P, such that $\langle P \cdot i \rangle$ is also a prefix of chain C.

4.1 Results

The following upper bounds are designed such that parties forward any message they receive each other and wait for as long as they can (or nearly as much as they can). By this we mean that parties only decide on values if the messages they received could have been all messages nonfaulty parties ever send throughout an execution of the protocol. The protocols are also conservative in the sense that parties default to outputting $\perp$ unless doing so might lead to a validity violation. A party is forced to output a value $x \neq \perp$ if its view could have been obtained in an execution in which all nonfaulty parties have the input x.

The protocol described in Algorithm 1 is designed to work as described above. Parties start by sending their signed input to all parties, and then forwarding that input to all parties. Whenever a party receives a signed input message it forwards that message to all parties. Every party p_i then waits until there are three parties (including itself) such that p_i received all of these parties' inputs, and the messages forwarding each other's inputs. Once that happens, p_i chooses whether to output the value x that it received as input, or the value $\perp$. If p_i saw that more than one party reported its input as $1 - x$ (either by receiving its input directly, or by receiving a forwarded input), p_i outputs $\perp$. Otherwise, p_i outputs x. We prove this protocol is a binding crusader agreement protocol in the full version of this paper.

Similarly to the previous protocol, in the protocol described in Algorithm 2, parties start by sending each other their inputs. They then forward any received input and any message forwarding an input, also indicating the messages' senders. Every party p_i then waits until there are three parties (including himself) that report consistent information about each other's messages. More specifically, they forward the same messages about each other as the messages the p_i received and forwarded. Then, p_i outputs its input x if it forwarded at most one input message with the value $1 - x$ and at most one of the three aforementioned parties forwarded more than one input message with the value $1 - x$. Otherwise, p_i outputs $\perp$.

We show that the protocol is a CA protocol for any number of parties n such that $n \geq 3f + 1$ in the full version of this work. We then proceed to show that the protocol is also binding for $n = 4, f = 1$ and $n = 7, f \geq 2$ in the full version of the paper, meaning that in these cases it is also a BCA protocol.

Algorithm 1 4-party authenticated Asynchronous BCA for Byzantine faults for party p_i.

Input: x

$fwdVals_1 = fwdVals_2 = fwdVals_3 = fwdVals_4 = \{\}$, $initVals = \{\}$
send $\langle i_x \rangle$ and $\langle i_x \cdot i \rangle$ to all, $fwdVals_i = fwdVals_i \cup \{i_x\}$
upon receiving $\langle k_v \rangle$ from p_k and not having forwarded a message from p_k:
  send $\langle k_v \cdot i \rangle$ to all
  $fwdVals_i = fwdVals_i \cup \{k_v\}$
  $initVals = initVals \cup \{k_v\}$
upon receiving $\langle j_v \cdot k \rangle$ from p_k
  $initVals = initVals \cup \{j_v\}$
  if j_{1-v} hasn't been added to $fwdVals_k$: $fwdVals_k = fwdVals_k \cup \{j_v\}$
upon $\exists p_j, p_k \neq p_i$ s.t. i_x, k_v, and $j_{v'}$ are in $fwdVals_i \cap fwdVals_k \cap fwdVals_j$ s.t. $v, v' \in \{0,1\}$:
  let S be the set $\{s | s_{1-x} \in initVals\}$
  if $|S| \leq 1$ **then** decide x
  else, decide $\perp$

Algorithm 2 7-party unauthenticated Asynchronous BCA for Byzantine faults for party p_i.

Input: x

$coreSet_i = \{\}$
for $j \in 1 \ldots n$:
  $initVals_j = \{\}$
  for $k \in 1 \ldots n$:
    $fwdedMsgs_{j,k} = []$
send $\langle i_x \rangle$ to all
upon receiving $\langle j_v \rangle$ from p_j and $fwdedMsgs_{i,j} = []$:
  send $\langle j_v \cdot i \rangle$ to all
  $initVals_i = initVals_i \cup \{j_v\}$
  $fwdedMsgs_{i,j} = fwdedMsgs_{i,j}.append(j_v)$
upon receiving $\langle k_v \cdot j \rangle$ from p_j and $k_* \cdot j \notin fwdedMsgs_{i,j}$:
  send $\langle k_v \cdot j \cdot i \rangle$ to all
  $initVals_j = initVals_j \cup \{k_v\}$
  $fwdedMsgs_{i,j} = fwdedMsgs_{i,j}.append(k_v \cdot j)$
  $fwdedMsgs_{j,k} = fwdedMsgs_{j,k}.append(k_v)$
upon receiving $\langle k_v \cdot l \cdot j \rangle$ from p_j and having received $k_v \cdot l$ from p_l:
  $fwdedMsgs_{j,l} = fwdedMsgs_{j,l}.append(k_v \cdot l)$
upon $\exists$ a set of $n - f$ distinct parties $coreSet_i$ s.t. the following 3 conditions hold:
1. $p_i \in coreSet_i$
2. $\forall (j,k,l) \in coreSet_i$, $fwdedMsgs_{j,k} = fwdedMsgs_{l,k}$
3. $\forall j \in coreSet_i$, $\exists v \in \{0,1\}$ s.t. $fwdedMsgs_{i,j}[1] = v_j$ and $\forall k \in coreSet_i$, $v_j \in initVals_k$

  $\forall j \in \{1 \ldots n\}$ let $S_j = \{s | s_{1-x} \in initVals_j\}$
  if $|S_i| \leq f$ and $|\{j \in \{1 \ldots n\} \text{ s.t. } |S_j| > f\}| \leq f$:
    decide x
  else decide $\perp$

References

1 Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 381–391, 2022. doi:10.1145/3519270.3538426.

2 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 599–616, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/aguilera.

3 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. Ubft: Microsecond-scale bft using disaggregated memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 862–877, 2023. doi:10.1145/3575693.3575732.

4 Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires $t + 1$ rounds. *Inf. Process. Lett.*, 71(3-4):155–158, 1999. doi:10.1016/S0020-0190(99)00100-3.

5 Hagit Attiya and Jennifer L. Welch. Multi-valued connected consensus: A new perspective on crusader agreement and adopt-commit, 2023. arXiv:2308.04646.

6 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*, pages 27–30, 1983. doi:10.1145/800221.806707.

7 Christian Cachin and Luca Zanolini. From symmetric to asymmetric asynchronous byzantine consensus. *arXiv preprint*, 2020. arXiv:2005.08795.

8 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, May 16-18, 1993, San Diego, CA, USA*, pages 42–51. ACM, 1993. doi:10.1145/167088.167105.

9 Tyler Crain. Two more algorithms for randomized signature-free asynchronous binary byzantine consensus with $t < n/3$ and $o(n^2)$ messages and $o(1)$ round expected termination. *arXiv preprint*, 2020. arXiv:2002.08765.

10 Richard A. DeMillo, Nancy A. Lynch, and Michael Merritt. Cryptographic protocols. In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 383–400. ACM, 1982. doi:10.1145/800070.802214.

11 Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 137–146. IEEE, 2006. doi:10.1109/DSN.2006.55.

12 Danny Dolev. The byzantine generals strike again. *Journal of algorithms*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

13 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.

14 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983. doi:10.1137/0212045.

15 Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a byzantine environment: Crash failures. *Inf. Comput.*, 88(2):156–186, 1990. doi:10.1016/0890-5401(90)90014-9.

16 Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982. doi:10.1016/0020-0190(82)90033-3.

17 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985. doi:10.1145/3149.214121.

18 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.

19 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019. doi:10.1109/DSN.2019.00063.

20 Leslie Lamport. Lower bounds on consensus. *Unpublished manuscript*, 2000.

21 Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006. doi:10.1007/S00446-006-0005-X.

22 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Comput.*, 19(2):104–125, 2006. doi:10.1007/S00446-006-0155-X.

23 Nancy Lynch. A hundred impossibility proofs for distributed computing. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 1–28, 1989. doi:10.1145/72981.72982.

24 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

25 Achour Mostéfaoui, 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*, pages 2–9, 2014. doi:10.1145/2611462.2611468.

26 Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: Multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020. doi:10.1145/3342195.3387529.

27 Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017. doi:10.1145/3127479.3128609.

28 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.

29 Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018. doi:10.1145/3269981.

A Proofs of Lower Bounds

▶ **Lemma 12.** *In* ***World 2*** *from the proof of Theorem 2, p_3 must decide (0, g) for $g \in \{1, 2\}$ without waiting for any round 2 messages from p_2.*

Proof.

World 2.a) (0-validity, maximally patient for p_1 crash, symmetric to World 1.a):

- All three parties have the input 0. p_2 and p_3 are nonfaulty, and p_1 crashes prior to sending any messages.
- p_2 and p_3 must decide (0, 2) without waiting for any messages dependent on p_1 by validity and termination.

World 2.b) (maximally patient for p_2 crash, symmetric to World 1.b):

- p_1 has input 1, while p_2 and p_3 start with input 0. p_2 sends round 1 messages, which are delivered to both p_1 and p_3, and then p_2 crashes. Following that, p_3's round 1 messages are delivered to p_1. Finally, p_1 and p_3's messages are delivered to each other without delay.

- Because p_2 crashed, p_1 and p_3 must decide without waiting for any additional messages from p_2, by termination.

We now argue why p_3 must decide $(0, g)$ for $g \in \{1, 2\}$ in **World 2** without waiting for any of p_2's round 2 messages. First, we show that p_2 decides (0, 2). Since p_1's messages are initially delayed, p_2 decides $(0, 2)$ due to indistinguishability from **World 2.a**, in which p_1 crashes. As a result, if p_3 decides, it must decide $(0, g)$ such that $g \in \{1, 2\}$ so as not to violate graded agreement. Next, we show why p_3 decides without waiting for any round 2 messages from p_2. This follows an indistinguishability argument with **World 2.b** for p_3, since any messages from p_2 apart from its round 1 messages are delayed for p_3 in **World 2**. ◀

▶ **Lemma 13.** *In **World 3** from the proof of Theorem 2, p_1 must output without waiting for any messages that depend on p_3.*

Proof.
World 3.a) (maximally patient for p_3 crash):

- p_1 and p_2 are nonfaulty, while p_3 crashes immediately before sending any messages. p_1 has input 1 and p_2 has input 0.
- p_1 and p_2 must decide without waiting for any messages dependent on p_3 by termination.

The lemma follows from a straightforward indistinguishability argument from **World 3.a)**, as any messages from p_3 and dependent on p_3 are delayed for p_1 in **World 3**. ◀

B Generalizing the Lower Bounds

In this section, we generalize the lower bounds from lower bounds specifically for $n = 3, n = 4$ or $n = 7$ to lower bounds for $n \geq 3, n \geq 4$ or $n \geq 7$. The techniques for generalizing the lower bound in the case that $n \geq 3, n \geq 4$ are standard and provided for completeness. On the other hand, generalizing the lower bound for $n \geq 7$ is slightly more intricate. In the following we simply show how to generalize two of the lower bounds presented above, but generalizing the other ones (with different corruption models or numbers of rounds) is done in the same manner.

We start by showing how to generalize the lower bound for $n = 4$ and $f = 1$ to any n, f such that $4f \geq n \geq 3f + 1$. Identical arguments can be made to generalize the lower bounds for $n = 3$ and $f = 1$ to any n, f such that $3f \geq n \geq 2f + 1$.

▶ **Theorem 14.** *Assume that it is impossible to solve Byzantine fault tolerant crusader agreement in two rounds with $n = 4$ parties and $f = 1$ faults. Then it is impossible to construct such a protocol for any $f \in \mathbb{N}$ and $4f \geq n \geq 3f + 1$.*

Proof. Assume by way of contradiction, that for some f, n such that $4f \geq n > 3f$ there exists a Byzantine fault tolerant crusader agreement protocol for n parties resilient to f corruptions in which all parties decide on a value after at most two rounds without a PKI setup. We will use this protocol to construct a Byzantine fault tolerant crusader agreement protocol for 4 parties with 1 corruption that requires the same number of rounds, contradicting the theorem statement.

The protocol is designed for 4 parties $p'_1, \ldots, p'_4$ which simulate a full run of the n-party protocol running with parties $p_1, \ldots, p_4$. Start by partitioning the parties $p_1, \ldots, p_n$ into 4 roughly-equal groups: $P_1, \ldots, P_4$. Since n is not necessarily a multiple of 4, it is possible that some of the groups will contain one more party than the other groups. More precisely, set $\ell = (n \mod 4)$, and let $P_1, \ldots, P_\ell$ be of size $\lceil \frac{n}{4} \rceil$ and $P_{\ell+1}, \ldots P_4$ be of size $\lfloor \frac{n}{4} \rfloor$. In case

that $\ell = 0$, this means that all set are exactly of size $\frac{n}{4}$. Note that in all other cases, this means that the sets do indeed contain a total of n parties, since their combined sizes are $\ell \cdot \lceil \frac{n}{4} \rceil + (4-\ell)\lfloor \frac{n}{4} \rfloor = \ell \cdot (\lfloor \frac{n}{4} \rfloor + 1) + (4-\ell)\lfloor \frac{n}{4} \rfloor = 4 \cdot \lfloor \frac{n}{4} \rfloor + (n \mod 4) = n$.

Now, in the 4-party protocol each party p'_i simulates the full n-party protocol for the parties in P_i. Every party p'_i receives an input x_i and simulates the actions of all parties in P_i after starting with the input x_i. This is done by running the code of each of those parties after receiving that input, and sending messages if required as described below. Whenever p'_i sees that party $p \in P_i$ sends a message m to some party $q \in P_j$ it does the following: if $j = i$, it simulates q receiving m by running the code that q would have run upon receiving the message from p. Otherwise, p'_i sends the message m to p'_j, along with the information that p sent the message to q. Similarly, when a party p'_j receives a message m from p'_i with the information that $p \in P_i$ sent that message to $q \in P_j$, p'_j simulates q receiving that message by running the code that q would have run upon receiving that message from p. Once p'_i sees that all of the simulated parties in P_i output values, it does the following: if at least one party in P_i output $\perp$, it outputs $\perp$. Otherwise, it outputs some non-$\perp$ value that a party in P_i output[2]. In this setting, the adversary can only corrupt a single party p'_i, which simulates the parties in P_i. The number of parties in P_i is at most $\lceil \frac{n}{4} \rceil$. By assumption, $n \leq 4f$, so $\lceil \frac{n}{4} \rceil \leq \lceil \frac{4f}{4} \rceil = f$. All other simulated parties act exactly the same as they would when receiving messages in the original protocol, since they are instructed to send and receive messages exactly as they would in the original protocol. In other words, the simulated run perfectly corresponds to a run in which the adversary corrupts at most f parties, in which messages between parties in the same set P_i are delivered immediately and the rest of the messages are delivered according to the scheduling dictated by the adversary. The protocol is secure under these conditions, and thus Validity, Agreement and Termination hold in the simulated run.

In order to complete the proof, all that is left to show is that the resulting 4-party protocol is a two-round Byzantine fault tolerant crusader agreement protocol with $n = 4$ and $f = 1$, reaching a contradiction to the theorem statement.

Validity. If all parties have the same input b, then each nonfaulty p'_i simulates all of the parties in P_i with the input b. This means that the run corresponds to a run in which all parties simulated by nonfaulty parties have the input b. From the Validity property of the original protocol, all simulated nonfaulty parties output b as well, and thus every nonfaulty p'_i output b after seeing that all of the parties in P_i output that value.

Agreement. Assume that two nonfaulty parties p'_i and p'_j output the non-$\perp$ value b_i and b_j respectively. Before doing so, each one saw that all of the parties simulated by it completed the protocol and that at least one of the parties simulated by p'_i and p'_j output b_i and b_j respectively. Those parties are simulated as nonfaulty parties, so $b_i = b_j$ from the Agreement property of the original protocol.

Termination. If each nonfaulty p'_i starts the protocol, it simulates all of the parties in P_i correctly throughout the whole protocol. This means that all of the parties in the P_i sets simulated by nonfaulty parties act as nonfaulty parties would in the original protocol, and thus eventually decide. After seeing that all of the parties in P_i output some value, every nonfaulty p'_i outputs a value as well.

Round Complexity. In the original n-party protocol, all parties output a value after two rounds. More precisely, all nonfaulty parties send only round 1 or round 2 messages. Observe a given run of the 4-party protocol. In the simulated n-party protocol, all

[2] An alternative choice is to output $\perp$ only if all simulated parties did, and otherwise output some non-$\perp$ value.

simulated parties output a value after at most 2 rounds without sending any message from round 3 or higher. Therefore, in the 4-party protocol, no party sends a message from round 3 message or higher, and after every nonfaulty simulated party decides a value, every nonfautly p'_i outputs a value as well. ◀

▶ **Theorem 15.** *Assume there is a network of 7 parties $p_1, \ldots, p_7$, and let S, T be a partitioning of the parties such that $|S| = 4, |T| = 3$, $S \cup T = \{p_1, \ldots, p_7\}$ and $S \cap T = \emptyset$. Assume that it is impossible to solve Byzantine fault tolerant binding crusader agreement in two rounds with $n = 7$ parties and $f = 2$ faults, even if the adversary can corrupt at most one party in S and one party in T. Then it is impossible to construct such a protocol for any $f \geq 2$ and $4f > n > 3f$.*

Proof. Assume by way of contradiction that such a protocol exists for some n, f such that $f \geq 2$ and $4f > n > 3f$. The proof follows a similar outline to the previous proof, simulating the n party protocol in the 7 party setting. Without loss of generality, assume that $S = \{p_1, \ldots, p_4\}$ and that $T = \{p_5, \ldots, p_7\}$. Since $4f > n > 3f$, there exists some $k \in [f - 1]$ such that $n = 3f + k$.

We will now construct a protocol for 7 parties $p'_1, \ldots, p'_7$. Start by partitioning the parties $\{p_1, \ldots, p_n\}$ into 7 sets $P_1, \ldots, P_7$. Each set in $P_1, \ldots, P_4$ contains k parties for the k defined above, and each party in $P_5, \ldots, P_7$ contains $f - k$ parties such that for every $i \neq j$, $P_i \cap P_j = \emptyset$. First, note that by definition $f > k > 0$ and thus also $f > f - k > 0$. This means that each of these sets has a positive number of parties, smaller than f. In addition, the total number of parties is $4 \cdot k + 3 \cdot (f - k) = 3f - 3k + 4k = 3f + k = n$. In other words, it is possible to partition the n parties into non-intersecting sets of these exact sizes.

From this point on, the simulation is exactly the same as in Theorem 14. Each party p'_i is in charge of simulating the parties in P_i. It starts the protocol by receiving its input x_i and simulating all of the parties in P_i starting the protocol with the same input x_i. Following that, if some simulated party $p \in P_i$ sends a message m to $q \in P_j$ it either delivers it immediately if $i = j$ or sends m to p'_j and signifies that p sent the message to q. Upon p'_j receiving a message m from p'_i saying that p sent that message to q, p'_j checks that $p \in P_i$ and $q \in P_j$. If that is the case, p'_j simulates q receiving that message from p. In all of the above discussion, by "simulating receiving the message" we mean that the simulating party runs the code that the simulated party would have run, and sends any messages according to the above description.

Once p'_i sees that all of the parties in P_i output some value, it outputs if at least one of the parties in P_i output $\perp$, p'_i outputs $\perp$ as well. Otherwise, it outputs some non-$\perp$ value that a party in P_i output. All that is left to do, is to show that the protocol is a 2-round protocol, resilient against a Byzantine adversary that controls at most one party in S and one party in T, reaching a contradiction. An adversary controlling at most one party in S and one party in T is in charge of simulating at most $f - k + k = f$ parties. This means that any run of the 7-party protocol corresponds to a run of the n-party protocol in which the adversary controls at most f parties, and the scheduling is the same as the one described in Theorem 14. Therefore, the simulated run terminates in two rounds and has the Validity, Agreement, Termination and Binding properties.

The proof that the 7-party protocol requires two rounds and that it has the Validity, Agreement and Termination properties is identical to the proof in Theorem 14 and is thus omitted. For the final property, Binding, assume some nonfaulty party p'_i outputs some value. At that point in time, it saw that all of the parties in P_i output values. All of those parties are nonfaulty, and thus from the Binding property of the n-party protocol, at that time there exists some value $b \in \{0, 1\}$ such that all nonfaulty parties output either b or $\perp$ in the

n-party protocol. We will show that all nonfaulty parties output either b or $\perp$ in the 7-party protocol. Observe some nonfaulty party p'_j in the 7-party protocol. If it outputs the value $\perp$ from the protocol, the property holds. Otherwise, it output some value b' after seeing that at least one party $p \in P_j$ output b', and no party in P_j output $\perp$. From the Binding property of the n-party protocol, $b' = b$, and thus p'_j outputs b as well. ◀

Distributed Partial Coloring via Gradual Rounding

Avinandan Das
Institut de Recherche en Informatique Fondamentale (IRIF),
CNRS and Université Paris Cité, France

Pierre Fraigniaud
Institut de Recherche en Informatique Fondamentale (IRIF),
CNRS and Université Paris Cité, France

Adi Rosén
Institut de Recherche en Informatique Fondamentale (IRIF),
CNRS and Université Paris Cité, France

Abstract

For $k \geq 0$, k-partial $(k+1)$-coloring asks to color the nodes of an n-node graph using a palette of $k+1$ colors such that every node v has at least $\min\{k, \deg(v)\}$ neighbors colored with colors different from its own color. Hence, proper $(\Delta+1)$-coloring is the special case of k-partial $(k+1)$-coloring when $k = \Delta$. Ghaffari and Kuhn [FOCS 2021] recently proved that there exists a deterministic distributed algorithm that solves proper $(\Delta+1)$-coloring of n-node graphs with maximum degree Δ in $O(\log n \cdot \log^2 \Delta)$ rounds under the LOCAL model of distributed computing. This breakthrough result is achieved via an original iterated rounding approach. Using the same technique, Ghaffari and Kuhn also showed that there exists a deterministic algorithm that solves proper $O(a)$-coloring of n-node graphs with arboricity a in $O(\log n \cdot \log^3 a)$ rounds. It directly follows from this latter result that k-partial $O(k)$-coloring can be solved deterministically in $O(\log n \cdot \log^3 k)$ rounds.

We develop an extension of the Ghaffari and Kuhn algorithm for proper $(\Delta+1)$-coloring, and show that it solves k-partial $(k+1)$-coloring, thus generalizing their main result. Our algorithm runs in $O(\log n \cdot \log^3 k)$ rounds, like the algorithm that follows from Ghaffari and Kuhn's algorithm for graphs with bounded arboricity, but uses only $k+1$ color, i.e., the smallest number c of colors such that every graph has a k-partial c-coloring. Like all the previously mentioned algorithms, our algorithm actually solves the general list-coloring version of the problem. Specifically, every node v receives as input an integer demand $d(v) \leq \deg(v)$, and a list of at least $d(v)+1$ colors. Every node must then output a color from its list such that the resulting coloring satisfies that every node v has at least $d(v)$ neighbors with colors different from its own. Our algorithm solves this problem in $O(\log n \cdot \log^3 k)$ rounds where $k = \max_v d(v)$. Moreover, in the specific case where all lists of colors given to the nodes as input share a common colors c^* known to all nodes, one can save one $\log k$ factor. In particular, for standard k-partial $(k+1)$-coloring, which corresponds to the case where all nodes are given the same list $\{1, \ldots, k+1\}$, one can modify our algorithm so that it runs in $O(\log n \cdot \log^2 k)$ rounds, and thus matches the complexity of Ghaffari and Kuhn's algorithm for $(\Delta+1)$-coloring for $k = \Delta$.

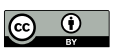
2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed graph coloring, partial coloring, weak coloring

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.30

Funding *Avinandan Das*: Additional support from ERC Consolidator Grant Distributed Biological Algorithms.
Pierre Fraigniaud: Additional support from the ANR Project ANR-20-CE48-0006 (DUCAT).
Adi Rosén: Most of the work of this author was done while with CNRS, FILOFOCS, Israel.

Acknowledgements We thank Alkida Balliu and Dennis Olivetti for useful discussions, and for pointing to us an error in a first draft of our result. We also thank Baruch Schieber for useful discussion in the early stage of this work. We finally thank the anonymous reviewers for their detailed comments.



1 Introduction

1.1 Partial Coloring

Proper coloring of the vertices of an arbitrary graph G with maximum degree Δ using $\Delta+1$ colors is one of the most studied symmetry-breaking problems in the context of distributed computing in networks [3, 17]. A natural generalization of proper coloring is *partial* coloring [2, 5, 11, 15]. Given two integers $k \geq 0$ and $c \geq 1$, k-partial c-coloring asks for a coloring of the vertices with colors in $\{1,\ldots,c\}$, such that every vertex v has at least $\min\{k, \deg(v)\}$ neighbors with a color different from its own color. In particular, k-partial $(k+1)$-coloring is equal to proper $(\Delta+1)$-coloring for $k=\Delta$, 0-partial coloring is trivial, and, for $0<k<\Delta$, k-partial $(k+1)$-coloring relaxes the requirement of proper coloring for vertices of degree larger than k. The case $k=1$ is referred to as *weak* coloring in [15].

Note that for every integer $k \geq 0$ there exists a k-partial $(k+1)$-coloring of G. Such a coloring can be constructed by a simple centralized greedy algorithm as follows. Initialize all vertices with color 1, and consider all vertices sequentially, one by one. For each considered vertex v, let $C(v)$ be the set of colors present in the neighborhood of v. If $|C(v)| = k+1$ then v keeps color 1, otherwise it recolors itself with an arbitrary color in $\{1,\ldots,k+1\} \setminus C(v)$. The resulting coloring is k-partial because (1) node v has at least $\min\{k, \deg(v)\}$ neighbors with a color different from its own color when it adopts its final color, and (2) two neighboring vertices with different colors at some time t during the execution of the greedy algorithm will remain with different colors at any time $t' \geq t$. This paper studies the design of a *distributed, deterministic* algorithm for k-partial $(k+1)$-coloring, that works for every $k \geq 0$.

We consider the standard LOCAL model of distributed computing [13]. This model assumes an n-node network modeled as a graph $G=(V,E)$, where each node is a computer, and the nodes communicate by exchanging messages along the edges of the graph. Each node is assigned an identifier in $\{1,\ldots,n^c\}$, for some $c \geq 1$, which is unique in the network. Initially, every node knows solely its identifier, the range of identifiers, and potentially some inputs, e.g., a non-negative integer k in the case of k-partial coloring. All nodes execute the same algorithm, which proceeds in synchronous rounds. At each round, every node sends one message to each of its neighbors, receives the messages from its neighbors, and performs some individual computation. After a certain number of rounds, every node outputs, and terminates. In the case of coloring, every node must eventually output a color in a prescribed range of colors, e.g., $\{1,\ldots,k+1\}$.

1.2 Previous Work

For many years, the best known (deterministic) algorithms for proper $(\Delta+1)$-coloring graphs with maximum degree Δ were running in essentially $2^{O(\sqrt{\log n})}$ rounds in n-node networks [1, 16]. This was the state of the art for almost a quarter of a century, and it is only a few years ago that an algorithm for $(\Delta+1)$-coloring running in a polylogarithmic number of rounds was proposed [18]. All these algorithms were based on a specific graph decomposition, and the efficient algorithm in [18] actually shows how to compute such a decomposition in a polylogarithmic number of rounds. Recently, a breakthrough result has been achieved, stating that $(\Delta+1)$-coloring can be solved in $O(\log n \cdot \log^2 \Delta)$ rounds [10], without using graph decomposition. Moreover, for large Δ, the algorithm sill runs in $O(\log^3 n)$ rounds – this has been very recently improved to $\tilde{O}(\log^2 n)$ rounds [8]. There are indeed algorithms for $(\Delta+1)$-coloring running in $\widetilde{O}(\sqrt{\Delta}) + O(\log^* n)$ rounds (see [4, 7, 14]). However, such algorithms are efficient for small Δ only, that is, their complexities are polylogarithmic in n only when Δ is itself growing polylogarithmically with n.

The algorithm in [10] is based on an original gradual rounding technique which, starting from a uniform selection of a color by each node, produces a non-necessarily proper coloring which leaves only a linear number of edges monochromatic (i.e., with both end-points having the same color). This technique has even been extended to computing a maximal independent set (MIS) [6], and to computing network decomposition [9]. In the present paper we show that the $O(\log n \cdot \log^2 \Delta)$-round $(\Delta+1)$-coloring algorithm from [10] can be modified to solve the generalized partial coloring problem. We note the gradual rounding technique can also be applied to coloring graphs with given *arboricity* a. Indeed, it was shown [10] that there exists a deterministic algorithm that solves proper $O(a)$-coloring of n-node graphs with arboricity a in $O(\log n \cdot \log^3 a)$ rounds. As a consequence, k-partial $O(k)$-coloring can be solved deterministically in $O(\log n \cdot \log^3 k)$ rounds: one can let every node v pick $\min\{k, \deg(v)\}$ incident edges arbitrarily, and then remove from the input graph G all edges that were not picked. The resulting graph G' has *degeneracy* at most $2k$ (simply because it has at most kn edges, and therefore minimum degree $2k$), and therefore it has arboricity at most $2k$ as well. By running in G' the algorithm from [10] for proper coloring graphs with bounded arboricity, we get that a k-partial $O(k)$-coloring of G', and therefore of G too, in $O(\log n \cdot \log^3 k)$ rounds. The question remains however whether one can reduce the number of colors from $O(k)$ to $k+1$. Indeed, for every $k \geq 1$, $k+1$ is the smallest number c of colors such that every graph has a k-partial c-coloring.

Note that the coloring algorithms in [10] actually solve the general list-coloring versions of the various problems. In the proper c-list-coloring problem, every node is also given a list of at least c colors as input, and its output color is bounded to belong to its list. We also slve the list-coloring version of partial coloring. In addition, we allow the *demand* of the nodes to vary, i.e., every node can have a different demand regarding how many of its neighbors must have a different color than its own.

Under restricted hypotheses, fast partial coloring algorithms exists. For instance, in regular graphs, there is a $O(\log^\star n)$-round deterministic algorithm for k-partial 3-coloring whenever $\Delta \geq 3k-4$ and $k \geq 3$, and for k-partial k-coloring whenever $\Delta \geq k+2$ and $k \geq 4$ (see [2]). Moreover, the same paper shows that computing 2-partial 2-coloring in Δ-regular graphs requires $\Omega(\log n)$ rounds deterministically, and $\Omega(\log \log n)$ rounds randomized, for any $\Delta \geq 2$. More generally, computing a k-partial c-coloring in Δ-regular graphs with $k \geq \frac{\Delta(c-1)}{c}+1$ requires $\Omega(\log n)$ rounds deterministic, and $\Omega(\log \log n)$ rounds randomized [2]. It was recently shown [5] that 1-partial 2-coloring in odd-degree graphs cannot be solved in $o(\log^\star \Delta)$ rounds, thereby providing a matching lower bound to the 30-year old upper bound in [15]. Finally, k-partial $O(k^2)$-coloring as well as d-defective $O(\Delta^2/d^2)$-coloring can both be computed in $O(\log^* n)$ rounds [11].

1.3 Our Results

For a non-negative integer k, the k-partial list-coloring problem in a graph $G=(V,E)$ is defined as follows.

Input of node v: A demand $d(v) \in \mathbb{N}$ satisfying $d(v) \leq \deg(v)$, a list $L(v)$ of colors, of size at least $d(v)+1$, and the value $k = \max_{v \in V} d(v)$.

Output of node v: A color $\delta(v) \in L(v)$ such that

$$|\{u \in V : \{u,v\} \in E \wedge \delta(v) \neq \delta(u)\}| \geq d(v)$$

.

The k-partial $(k+1)$-coloring problem is the special case of k-partial list-coloring problem in which every node has demand $\min\{k, \deg(v)\}$, and all lists are all equal to $\{1, \ldots, k+1\}$. Our main results are the following.

▶ **Theorem 1.** *There exists a deterministic distributed algorithm solving k-partial list-coloring in all n-node networks, and running in $O(\log n \cdot \log^3 k)$ rounds under the* LOCAL *model.*

Note that the round-complexity of our algorithm scales as a function of the upper bound k on the demands, and not as a function of Δ. In particular, for $k = O(1)$, our algorithm runs in just $O(\log n)$ rounds, even in graphs with large maximum degrees. Also note that even if it is possible to design faster algorithms for specific values of the demand, e.g., for 1-partial 2-coloring, a.k.a. *weak coloring* [15], our algorithm is generic, and works for all $k \geq 0$.

If the lists given to the nodes satisfy $\cap_{v \in V} L(v) \neq \varnothing$, and the nodes are given a color $c^* \in \cap_{v \in V} L(v) \neq \varnothing$ as input, one can modify our algorithm so as to solve k-partial list-coloring in $O(\log n \cdot \log^2 k)$ rounds. An important application is k-partial $(k+1)$-coloring in which every node has demand $\min\{k, \deg(v)\}$, and all lists are all equal to $\{1, \ldots, k+1\}$. Indeed, $1 \in \cap_{v \in V} L(v)$ in k-partial $(k+1)$-coloring, and this holds even if, for every node v, $L(v) = \{1, \ldots, d(v)+1\}$ for a demand $d(v)$.

▶ **Theorem 2.** *There exists a deterministic distributed algorithm solving k-partial $(k+1)$-coloring in all n-node networks, and running in $O(\log n \cdot \log^2 k)$ rounds under the* LOCAL *model.*

1.4 Our Techniques

We base ourselves on the main ideas of the algorithm by Ghaffari and Kuhn [10] for $(\Delta+1)$-coloring, which actually works for list-coloring as well, and on the core rounding procedure of that algorithm. The algorithm in [10] proceeds in $O(\log n)$ iterations of a procedure whose objective is to fix the color of a constant fraction of the remaining uncolored nodes. Specifically, at each iteration, if $G_i = (V_i, E_i)$ denotes the subgraph of the input graph G whose nodes are still uncolored at the ith iteration, the procedure starts from a fractional coloring of G_i, where each color $c \in L_i(v)$ in the current list of colors assigned to every node $v \in V_i$ appears with "weight" $1/|L_i(v)|$. Given this fractional coloring, the procedure produces, in $O(\log^2 \Delta)$ rounds, an integral coloring γ_i of the nodes of G_i, that is, for every $v \in V_i$, all colors $c \in L_i(v)$ but one have weight 0, and the remaining color has weight 1. This coloring γ_i is not necessarily proper. However, it is shown (see Corollary 3.6 in [10]) that the total number of monochromatic edges F_i in E_i is $O(|V_i|)$. As a consequence, a constant fraction of the nodes in the graph $G'_i = (V_i, F_i)$ have degrees upper bounded by a constant, from which it follows that a maximal independent set (MIS) I_i in the subgraph H_i of G'_i induced by these low degree nodes can be computed in $O(\log^\star n)$ rounds [12]. Moreover, since the maximum degree of H_i is bounded, there is a constant fraction of its nodes in I_i. The nodes in I_i adopt their colors given by γ_i, and terminate. The remaining nodes update their lists by removing the colors adopted by neighboring nodes that terminated, and carry on with yet another iteration. The algorithm in [10] uses other clever tricks, but the above summarizes the core of the algorithm, which is the part that we shall modify for handling partial coloring.

Our first change is in the preprocessing stage where we compute a k-partial $O(k^2)$-coloring α of the graph and then remove all monochromatic edges. Since the computed coloring α is k-partial, the removal of the monochromatic edges still leaves us with a graph on which a k-partial coloring is a k-partial coloring of the original graph. On the other hand we have a proper coloring of the remaining graph, a condition that is needed to apply the core procedure in [10].

At the start of each iteration i, a preprocessing phase selects arbitrarily, for each node, a number of "outgoing" edges equal to the size of this node's color list (or demand), and removes all edges that were not selected by any of their endpoints. The remaining graph $G_i = (V_i, E_i)$ has the property that a partial coloring on it is also a corresponding partial coloring on the original graph (for that iteration), and the (remaining) degree of each node is not bigger that the size of its (remaining) list of colors. The rounding procedure of Ghaffari and Kuhn [10] is applied on G_i, providing each node $v \in V_i$ with a color $\gamma_i(v) \in L_i(v)$. We then introduce the notion of *saturated* nodes w.r.t. γ_i, which are nodes v such that, for each color c in their lists, they have a neighbor u_c colored c by γ_i. These nodes might get "saturated" with respect to the following consideration. Let S_i denotes the set of saturated nodes, and let $v \in S_i$. If a neighbor u_c of v, colored c, belongs to the MIS computed at iteration i, it is expected that u_c terminates with color c, and v removes c from its list $L_i(v)$ so that to eventually provide u_c with a neighbor with a color different from c. But if v is saturated, this would result in exhausting all possible colors in the list of $L_i(v)$, preventing v from eventually choosing a color during further iterations.

To overcome the issue of saturated nodes, we proceed differently from [10], by first letting all nodes that are saturated w.r.t. γ_i adopting their current color, and terminate. Then, we also introduce the notion of *idle* nodes. Roughly, an idle node is a node which is not saturated, but which has so many saturated neighbors with potentially the same color as its own color that it has not sufficiently many other neighbors for guaranteeing that its demand will eventually be satisfied. In our algorithm, idle nodes abort the search for a color during the remaining iterations. They do not participate to the subsequent iterations, but update their demand and list of colors according to the colors picked by neighboring (active) nodes. Finally, if K_i denotes the set of nodes that become idle at the ith iteration, then all nodes in $I_i \setminus (S_i \cup K_i)$, that is, all nodes in the MIS that are neither saturated nor idle, adopt their current colors. In this way, k-partial coloring is guaranteed for the non-idle nodes, and each of the $O(\log n)$ iterations performs in $O(\log^2 k)$ rounds.

After all iterations are completed, it remains to assign colors to the nodes which became idle during some of the iterations, during a post-processing stage. We note that the subgraph induced by the idle nodes can be viewed as "layered", where the nodes that became idle at iteration i form layer K_i of the subgraph, $i = 1, \ldots, O(\log n)$. We then show that the preconditions required for applying Lemma 5.4 in [10], which provides an algorithm for proper coloring layered graphs, are fulfilled (in particular, the degree towards the upper layers is bounded by k). Indeed, it occurs that the threshold for the number of saturated neighbors, which defines idle nodes, precisely corresponds to the ability to properly color the idle nodes. By application of Lemma 5.4 in [10], this post-processing stage runs in $O(\log n \cdot \log^3 k)$ rounds. Note that, up to the post-processing stage, our algorithm runs in $O(\log n \cdot \log^2 k)$ rounds as each of the $O(\log n)$ iterations consumes $O(\log^2 k)$ rounds, and the extra $\log k$ factor is only due to the post-processing stage.

Partial Coloring

Assuming that all lists contain a common color c^*, which fits with the setting of k-partial $(k+1)$-coloring, enables to simplify the coloring procedure performed at each of the $O(\log n)$ iterations of the algorithm, and to avoid the costly post-processing stage. Roughly, we can avoid introducing the notion of idle nodes as follows. The color c^* is removed from all lists, and will play the role of a backup color in case a node has exhausted all the colors in its list for satisfying neighbors. Specifically, as in [10], we serve first all the nodes in the maximal independent set (MIS) I_i computed at iteration i, i.e., for every $v \in I_i$, node v adopts its

color $\gamma_i(v)$. Then, if a node v gets saturated (i.e., all the colors in its list are picked by at least one of its neighbor, which also happens to be in the MIS), then v adopts color c^* – recall that c^* was removed from all lists. Letting the saturated nodes adopting color c^* does not cause other nodes to saturate at further iterations even if it may seem that a node v with many neighbors that have fixed their color to c^* at previous iterations might be prevented from using color c^* itself (in case it becomes saturated). Indeed, we show that while the number of neighbors of a node v may decrease at a given iterations as these neighbors adopt color c^* and terminate, the set of colors currently in the list of that node v is not affected by these neighbors. As a consequence, we can show that if a node has "many" neighbors colored c^* then the size of its list actually becomes larger than its current degree, and thus this node cannot become saturated anymore. This guarantees the correctness of the algorithm. Finally, avoiding the post-processing stage enables saving one $\log\ k$ factor in the round complexity, since each iteration performs in $O(\log^2 k)$ rounds. Due to lack of space, the details are moved to Appendix A.

2 The Algorithm

We describe our distributed algorithm for solving partial list-coloring in an arbitrary graph. The algorithm is to be executed in the LOCAL model by the nodes of an n-node graph $G = (V, E)$. Due to lack of space, we do not re-explain some of the features of the algorithm in [10], but mostly focus on the parts that differ significantly from this latter algorithm for extending it from $(\Delta + 1)$-list coloring to k-partial list coloring. Our algorithm proceeds as follows.

2.1 Preprocessing Stage

The nodes compute a k-partial $O(k^2)$-coloring of G, which we denote by α. This can be done in $O(\log^* n)$ rounds (see [11]). Then, the nodes remove the monochromatic edges to obtain a sub-graph of G, denoted by $G_1 = (V_1, E_1)$, where $V_1 = V$. Note that in G_1 the degree of any node $v \in V$ is at least $\min\{k, \deg_G(v)\}$, and therefore $d(v) \leq \deg_{G_1}(v)$. Note also that α is a proper $O(k^2)$-coloring of G_1.

2.2 Core of the Algorithm

The coloring produced by our algorithm is denoted $\delta(\cdot)$. Initially, $\delta(v) = \perp$ for every $v \in V$. For computing their colors, the nodes perform $O(\log n)$ iterations of a series of phases, called preprocessing, derandomization, color assignment, and update. At each iteration, some nodes $v \in V$ fix their colors $\delta(v) \in L(v)$. Once a node v becomes colored (i.e., it adopts a color $\delta(v) \neq \perp$), its color never changes. At each iteration, we classify the nodes of G into three categories: *active, idle*, and *terminated*.

- A terminated node v is a node that has adopted its color $\delta(v)$.
- An idle node is a node that has not yet adopted its color, but it stops participating in the subsequent iterations after it became idle. However, its demand, as well as its list of colors are updated in each subsequent iteration.
- An active node is a node that is neither idle nor terminated (such a node will carry on to the next iteration).

An active node may, in a subsequent iteration, be colored (in which case it becomes terminated), or become idle. Only active nodes proceed with another iteration. An idle node remains idle until the post-processing stage starts[1]. When this is the case, no more iterations are performed, and the idle nodes perform a specific post-processing stage of computation for finalizing their colors (this stage to be described further in the text, in Section 2.3). Once this is done, the algorithm terminates. Initially, i.e., before the first iteration starts, all nodes are active.

We now describe one iteration of the algorithm. The input to iteration $i \geq 1$ is a graph $G_i = (V_i, E_i)$, a demand function $d_i : V_i \to \mathbb{N}$, and a color list function $L_i : V_i \to 2^{\mathbb{N}}$. If, for a node v still active at iteration i, $|L_i(v)| > d_i(v) + 1$, then v prunes its list arbitrarily for keeping exactly $d_i(v) + 1$ colors in its list. For every node v, we set $d_1(v)$ as the input demand $d(v)$, and $L_1(v)$ as the input list $L(v)$.

2.2.1 Preprocessing Phase

Every vertex v arbitrarily selects $d_i(v)$ many edges incident to it – we shall show later, in the proof of correctness, that v has sufficiently many edges for performing this selection. Note that an edge can be selected by one or two of its endpoints. Any non-selected edge is removed from G_i. In what follow, we slightly abuse notation, and $G_i = (V_i, E_i)$ still refers to the actual graph after the removal of the non-selected edges. Note that G_i has average degree at most $2k$, and therefore $O(kn)$ edges.

2.2.2 Derandomization Phase

Every vertex v picks an arbitrary set $L'_i(v) \subseteq L_i(v)$ of $2^{\lfloor \log |L_i(v)| \rfloor}$ colors. Observe that $\frac{|L_i(v)|}{2} \leq |L'_i(v)| \leq |L_i(v)|$. For any set $p = \{p_v : v \in V_i\}$ of random distributions over the lists $L'(v)$, $v \in V_i$, and for $L = \cup_{v \in V} L'_i(v)$, we define a weight function $w_p : (V_i \times L)^2 \to [0, 1]$ as follows:

$$w_p\big(\{(u, a), (v, b)\}\big) = \begin{cases} 0 & \text{if } \{u, v\} \notin E_i \\ 0 & \text{if } \{u, v\} \in E_i \text{ and } a \neq b \\ p_u(a) \cdot p_v(b) & \text{if } \{u, v\} \in E_i \text{ and } a = b \end{cases}$$

Moreover, we define

$$W(p) = \sum_{((u,a),(v,b)) \in (V_i \times L)^2} w_p\big((u, a), (v, b)\big).$$

Corollary 3.6 in [10] provides a way to "derandomize" the uniform distribution p^{unif}, defined as

$$p_v^{\text{unif}}(c) = \frac{1}{|L'_i(v)|}$$

for every node v and every color $c \in L'_i(v)$, while almost preserving $W(p)$. We use Corollary 3.6 in [10] with $\epsilon = 1$, using the proper $O(k^2)$-coloring α computed during the pre-processing stage (cf. Section 2.1), and observing that our node "labeling" uses at most $2^{\lfloor \log(k+1) \rfloor}$ colors.

[1] Note that since all nodes know $k = \max_{v \in V} d(v)$ and the (polynomial) range of IDs, they can individually compute how long lasts each phase, each iteration, and each stage of the algorithm. In particular, a node becoming idle at a given iteration knows when it has to start executing the post-processing stage.

▶ **Lemma 3** (Corollary 3.6 in [10]). *A coloring γ_i satisfying $\gamma_i(v) \in L'_i(v)$ for every $v \in V_i$ can be computed in $O(\log^2 k)$ rounds, such that $W(p^{\gamma_i}) \leq 2 \cdot W(p^{\text{unif}})$, where the distribution $p^{\gamma_i} = (p_v^{\gamma_i})_{v \in V_i}$ is defined as*

$$p_v^{\gamma_i}(c) = \begin{cases} 1 & \text{if } \gamma_i(v) = c \\ 0 & \text{otherwise .} \end{cases}$$

2.2.3 Color Assignment Phase

Given the graph $G_i = (V_i, E_i)$, and the coloring γ_i from Lemma 3, let us consider the graph $G'_i = (V_i, F_i)$ where $F_i \subseteq E_i$ is the set of monochromatic edges w.r.t γ_i. As we will prove later in the proof of correctness (see Lemma 10), we have $|F_i| \leq 4 \cdot |V_i|$. As a consequence, the number of nodes $v \in V_i$ with $\deg_{G'_i}(v) \geq 16$ is at most $|V_i|/2$. Let us denote by H_i the subgraph of G'_i induced by the nodes with degree less than 16 in G'_i. By construction, H_i has at least $|V_i|/2$ nodes (again, see Lemma 10), and it has maximum degree at most 15. The facts that H_i has bounded degree, and that it has a $O(k^2)$ proper coloring thanks to the pre-processing stage imply that its nodes can compute a maximal independent set (MIS) I_i in H_i in $O(\log^\star k)$ rounds [12] . Note that, since each of the at least $|V_i|/2$ nodes in H_i has maximum degree 15, we have $|I_i| \geq \frac{|V_i|}{32}$.

We now introduce a notion that plays an important role in our algorithm. For every node v, let $N_{G_i}(v)$ denote the set of v's neighbors in G_i.

▶ **Definition 4.** *A node $v \in V_i$ is* saturated *with respect to the coloring γ_i if, for every $c \in L_i(v)$, there exists a neighbor $u \in N_{G_i}(v)$ with $\gamma_i(u) = c$.*

Note that some nodes in the maximal independent set may be saturated, whereas some other nodes in the independent set may not be saturated, and the same holds for nodes outside the independent set. At this point, nodes may turn from active to idle or terminated according to one of the following three rules.

- **Saturated Node Rule:** Every saturated node w.r.t. γ_i fixes its final color as $\delta(v) = \gamma_i(v)$, and terminates. Let us denote by S_i the set of nodes that become saturated w.r.t. γ_i at the ith iteration.
- **Idle Node Rule:** For every node $v \in V_i \setminus S_i$, if

$$|N_{G_i}(v)| - |\{u \in N_{G_i}(v) \cap S_i \ : \ \gamma_i(u) = \gamma_i(v)\}| < d_i(v) \tag{1}$$

  then v becomes idle. Let us denote by K_i the set of nodes that become idle at the ith iteration.
- **MIS Node Rule:** Every vertex $v \in I_i \setminus (S_i \cup K_i)$ fixes its final color as $\delta(v) = \gamma_i(v)$, and terminates. We denote by J_i the set of non-saturated node nodes in the independent set that terminate at iteration i, i.e., $J_i = I_i \setminus (S_i \cup K_i)$.

Remark. The intuition guiding the above three rules is that a saturated node v has all colors in $L_i(v)$ present its neighborhood and hence it has at least $d_i(v)$ neighbors with colors, in γ_i, different than $\gamma_i(v)$. It can therefore safely set its final color to its current color: for each neighbor either that neighbor adopts its current (different) color, or will be "instructed" not to use v's color is subsequent iterations. On the other hand, nodes satisfying Eq. (1) are nodes that are in some sense "stuck" because they have too many neighbors with the same color as their own color, which do not leave them enough "space" for accommodating their demand. This is why the algorithm "freezes" them, as idle nodes, to be treated after the

termination of all iterations, during the post-processing stage (see Section 2.3). Finally, nodes in the MIS fix their colors. For making sure that its demand will be eventually satisfied, each MIS node will require its neighbors to remove its color from their lists (see the updating phase hereafter). Here comes the main reason of introducing saturated nodes. It may indeed be the case that a node v is surrounded by MIS nodes, with colors covering its list entirely. All these MIS nodes will prevent v from using any of its available colors, resulting in color starvation. For avoiding this, a saturated node fixes its color anyway, and terminates. We shall see that this does not prevent its adjacent MIS nodes to have their demands eventually satisfied. Essentially, the reason is that if they could not be satisfied, then they would have become idle.

2.2.4 Updating Phase

Towards the next iteration we perform a series of list and demand updates, applying to both the active *and* the idle nodes. Let us denote by K the entire set of idle nodes, i.e., $K = \bigcup_{j=1}^{i} K_j$.

List Update. The lists are updated as follows:

- For every active node v, i.e., for every $v \in V_i \setminus (S_i \cup K_i \cup J_i)$,

$$L_{i+1}(v) = L_i(v) \setminus \{\delta(u) \ : \ u \in N_{G_i}(v) \cap (S_i \cup J_i)\}.$$

- For every idle node v, i.e., for all $v \in K$ (and not only $v \in K_i$),

$$L_{i+1}(v) = L_i(v) \setminus \{\delta(u) \ : \ u \in N_{G_1}(v) \cap (S_i \cup J_i)\}.$$

  Note that for idle nodes we consider $u \in N_{G_1}(v)$, and not only $u \in N_{G_i}(v)$, since $v \in K$ may have became idle before iteration i, say at iteration $j < i$, i.e., $v \in K_j$, and thus v may not belong to V_i.

Demand Update. The demands are updated as follows:

- For every active node v, i.e., for every $v \in V_i \setminus (S_i \cup K_i \cup J_i)$,

$$d_{i+1}(v) = \max\{0, \ d_i(v) - |N_{G_i}(v) \cap (S_i \cup J_i \cup K_i)|\}.$$

- For every idle node v, i.e., for all $v \in K$ (and not only for those in K_i),

$$d_{i+1}(v) = \max\{0, \ d_i(v) - |N_{G_1}(v) \cap (S_i \cup J_i)|\}.$$

  Note again that we consider for idle nodes $N_{G_1}(v)$, and not only $N_{G_i}(v)$ for the same reasons as above.

Graph Update. The graph G_i is updated to $G_{i+1} = (V_{i+1}, E_{i+1})$ as follows:

- $V_{i+1} = V_i \setminus (S_i \cup J_i \cup K_i)$
- $E_{i+1} = \big\{\{u, v\} \in E_i \ : u \in V_{i+1} \wedge v \in V_{i+1}\big\}$

The nodes that remain active at the end of the iteration proceed to the next iteration.

2.3 Post-Processing Stage

After all active nodes eventually become either terminated (with their final color) or idle, it remains to provide the idle nodes with a color. Every idle node belongs to one of the set K_i, $1 \leq i \leq T$, where T is the number of iterations of the four phases described in Section 2.2 – we shall prove later that $T = O(\log n)$. Therefore, for $K = \cup_{i=1}^{T} K_i$, the graph $G_1[K]$ induced by the idle vertices can be viewed as *layered*, with layers $K_1, \ldots, K_T$. Every idle node has a list of available colors equal to $L_{T+1}(v)$, as computed during the updating phase of the last iteration T. We will apply Lemma 5.4 in [10] with $h = T$, $\hat{\Delta} = k$, and the lists $M(v) = L_{T+1}(v)$ for all $v \in K$. Let $i \in \{1, \ldots, T\}$, and let $v \in K_i$. We denote by $\deg^+(v)$ the number of neighbors of v belonging to layers $K_i, \ldots, K_T$.

▶ **Lemma 5** (Lemma 5.4 in [10]). *Let us consider a layered graph $G_1[K]$ with h layers $K_1, \ldots, K_h$ such that, for every node $v \in K$, v is assigned a list of colors $M(v)$ with $|M(v)| > \deg^+(v)$. Let $\hat{\Delta} = \max_{v \in K} \deg^+(v)$. There is a deterministic distributed algorithm that solves proper list-coloring with input lists M in $G[K]$, running in $O(\log n \log^2 \hat{\Delta} + h \log^3 \hat{\Delta})$ rounds.*

In the proof of correctness, we shall show that the conditions in Lemma 5 hold for the post-processing stage with $\hat{\Delta} = k$. Every idle node adopts the color returned by the proper list-coloring algorithm in Lemma 5.

This completes the description of our algorithm. In the next section, we shall prove its correctness.

3 The Proof

We first prove that our algorithm does solve k-partial list-coloring. Next, we will prove that it runs in the prescribed number of rounds.

3.1 Proof of Correctness

The lemma below shows the correctness of the preprocessing stage described in Section 2.1.

▶ **Lemma 6.** *The k-partial list-coloring problem is well defined on the graph $G_1 = (V_1, E_1)$. Moreover, any k-partial list-coloring on the graph $G_1 = (V_1, E_1)$ is also a k-partial list-coloring on the original graph $G = (V, E)$.*

Proof. To show that the k-partial list-coloring problem is well defined on the graph G_1, we must show that, for every node, its demand does not exceed its degree (in G_1). Recall that the graph $G_1 = (V_1, E_1)$ (i.e., the graph after the preprocessing stage) satisfies $V_1 = V$, so the demands and lists are defined on the same set of vertices. Let $v \in V$ be a vertex. We have $d(v) \leq \deg_G(v)$, $d(v) < |L(v)|$, and $d(v) \leq k = \max_{u \in V} d(u)$.

- If $\deg_G(v) \leq k$, then none of the edges incident to v in G is monochromatic in the k-partial coloring α of G computed in the preprocessing phase (see Section 2.1). It follows that $\deg_{G_1}(v) = \deg_G(v)$.
- If $\deg_G(v) > k$, then $\deg_{G_1}(v) \geq k$ as the coloring α computed in the preprocessing phase is a k-partial coloring of G.

Therefore, either $\deg_{G_1}(v) = \deg_G(v)$ or $\deg_{G_1}(v) \geq k$. As a consequence, $d(v) \leq \deg_{G_1}(v)$, as claimed. Now, let ψ be a k-partial list-coloring on the graph G_1. We have $\psi(v) \in L(v)$, and v has at least $d(v)$ neighbors in G_1 with another color. Since $E_1 \subseteq E$, v has also at least $d(v)$ neighbors in G with another color. Therefore ψ is a k-partial list-coloring on the original graph G. ◀

By Lemma 6, it is sufficient to prove the correctness of the algorithm on the graph G_1 (i.e., the graph after the preprocessing stage). Let us start by establishing a series of invariants which hold throughout the execution of the algorithm. The following holds by construction.

▶ **Observation 7.** *For every iteration $i \geq 1$, S_i, J_i, K_i, and V_{i+1} form a partition of V_i.*

The next lemma shows that the conditions necessary for partial list-coloring are preserved.

▶ **Lemma 8.** *For every iteration $i \geq 1$, and every node $v \in V_i$, we have that $d_i(v) \leq \deg_{G_i}(v)$, and $d_i(v) < |L_i(v)|$.*

Proof. We prove the lemma by induction on the iteration number, i. The basis of the induction holds for $i = 1$ by Lemma 6. Let us assume that the lemma holds for $i \geq 1$, and let $v \in V_{i+1}$. We have $V_{i+1} = V_i \setminus (S_i \cup J_i \cup K_i)$, and

$$d_{i+1}(v) = \max\{0,\ d_i(v) - |N_{G_i}(v) \cap (S_i \cup J_i \cup K_i)|\},$$

from which it follows that $d_{i+1}(v) \leq \deg_{G_{i+1}}(v)$. For the list update, we have

$$L_{i+1}(v) = L_i(v) \setminus \{\delta(u) : u \in N_{G_i}(v) \cap (S_i \cup J_i)\}.$$

Therefore, since

$$|\{\delta(u) : u \in N_{G_i}(v) \cap (S_i \cup J_i)\}| \leq |N_{G_i}(v) \cap (S_i \cup J_i \cup K_i)|,$$

we have $d_{i+1}(v) < |L_{i+1}(v)|$. Thus the lemma holds for $i+1$, which completes the proof. ◀

The idle nodes carry on updating their lists and demands after they become idle, but the invariant $d_i(v) < |L_i(v)|$ remains true for idle nodes too, as shown below.

▶ **Lemma 9.** *Let $i \geq 1$, and let $v \in K_i$. For every $j \geq i$, $d_j(v) < |L_j(v)|$.*

Proof. Let us fix $i \geq 1$. We prove the claim by induction on $j \geq i$. Since $v \in K_i$, the base case holds by Lemma 8, as $v \in V_i$. For the induction step, let us assume that $d_j(v) < |L_j(v)|$, for $j \geq i$. We have

$$L_{j+1}(v) = L_j(v) \setminus \{\delta(u) : u \in N_{G_1}(v) \cap (S_j \cup J_j)\}.$$

and

$$d_{j+1}(v) = \max\{0,\ d_j(v) - |N_{G_1}(v) \cap (S_j \cup J_j)|\},$$

Therefore, since

$$|\{\delta(u) : u \in N_{G_1}(v) \cap (S_j \cup J_j)\}| \leq |N_{G_1}(v) \cap (S_j \cup J_j)|,$$

we get that $d_{j+1}(v) < |L_{j+1}(v)|$, as desired. ◀

We now prove that the number of iterations performed during the core of the algorithm (see Section 2.2) is finite. We start with the following lemma that we obtain by modifying a similar proof from [10].

▶ **Lemma 10.** *Let $i \geq 1$. The graph $G'_i = (V_i, F_i)$ induced by the monochromatic edges in $G_i = (V_i, E_i)$ w.r.t. γ_i has a linear number of edges. Specifically, $|F_i| \leq 4\,|V_i|$.*

Proof. Let us analyze the total "weight" $W(p^{\text{unif}})$ of the uniform distribution p^{unif} before applying rounding (all notations except one are defined at the beginning of Section 2.2.2). We introduce just one new notation: $N_{G_i}^{select}(v)$ denotes the set of selected neighbours of v according to the operation performed in Section 2.2.1 where v selects $d_i(v)$ many edges incident to it. We have

$$\begin{aligned} W(p^{\text{unif}}) &= \sum_{(u,a),(v,b)\in(V_i\times L)^2} w_{p^{\text{unif}}}((u,a),(v,b)) \quad = \sum_{\substack{\{u,v\}\in E_i \\ c\in L_i'(u)\cap L_i'(v)}} p_u(c)\cdot p_v(c) \\ &= \sum_{\substack{\{u,v\}\in E_i \\ c\in L_i'(u)\cap L_i'(v)}} \frac{1}{|L_i'(u)|}\cdot\frac{1}{|L_i'(v)|} \quad = \sum_{\{u,v\}\in E_i} |L_i'(u)\cap L_i'(v)|\cdot\frac{1}{|L_i'(u)|}\cdot\frac{1}{|L_i'(v)|} \\ &\leq \sum_{v\in V_i}\sum_{u\in N_{G_i}^{select}(v)} |L_i'(u)\cap L_i'(v)|\cdot\frac{1}{|L_i'(u)|}\cdot\frac{1}{|L_i'(v)|} \quad \leq \sum_{v\in V_i}\sum_{u\in N_{G_i}^{select}(v)} \frac{1}{|L_i'(v)|}. \end{aligned}$$

Now, $|L_i'(v)| \geq d_i(v)/2$, and $|N_{G_i}^{select}(v)| = d_i(v)$. Thus

$$W(p^{\text{unif}}) \leq \sum_{v\in V_i}\sum_{u\in N_{G_i}^{select}(v)} \frac{1}{d_i(v)/2} = \sum_{v\in V_i}\frac{d_i(v)}{d_i(v)/2} = 2\,|V_i|.$$

By applying lemma 3, we have that $W(p^{\gamma_i}) \leq 4\,|V_i|$. It follows from the definition of p^{γ_i} that the size of the set F_i of monochromatic edges in G_i w.r.t. γ_i is at most $4\,|V_i|$. ◀

As a consequence of the previous lemma, the core of our algorithm terminates.

▶ **Lemma 11.** *The algorithm described in Section 2 terminates.*

Proof. Lemma 10 states that, for every $i \geq 1$, the subgraph $G_i' = (V_i, F_i)$ of $G_i = (V_i, E_i)$ satisfies $|F_i| \leq 4\,|V_i|$. Therefore, the subset $V_i' \subseteq V_i$ of the nodes $v \in V_i$ with $\deg_{G_i'}(v) \leq 16$ satisfies $|V_i'| \geq |V_i|/2$. As a consequence, the subgraph H_i of $G_i' = (V_i, F_i)$ induced by the nodes in V_i' has at least $|V_i|/2$ nodes. It follows that there is a non-empty maximal independent set in H_i, which guarantees that $I_i \neq \varnothing$. Therefore, either $S_i \neq \varnothing$, or $J_i = I_i \smallsetminus (S_i \cup K_i) \neq \varnothing$, which ensures that at least one vertex terminates at iteration i. As a consequence, the number of iterations performed during the core of the algorithm is at most n (we shall show later that it is actually at most $O(\log n)$). ◀

Let T be the number of iterations of the algorithm until there are no more active nodes left, and let us first focus on the status of the idle nodes at the end of the T iterations. For any idle node v, let $i \geq 1$ be the iteration during which v becomes idle, i.e., $v \in K_i$, and let us then denote by

$$\deg^+(v) = |N_{G_1}(v) \cap (K_i \cup \cdots \cup K_T)|,$$

the number of neighbors of v in G_1 that belong to the set $K_i \cup \cdots \cup K_T$. Note that the idle nodes update their demands and lists at the end of the Tth iteration, so $d_{T+1}(v)$ and $L_{T+1}(v)$ are well defined for an idle node $v \in K = \cup_{i=1}^{T} K_i$. The following justifies the use of Lemma 5 in the post-processing stage (see Section 2.3).

▶ **Lemma 12.** *For every $v \in K$, $\deg^+(v) < |L_{T+1}(v)|$ and $\deg^+(v) \leq k$.*

Proof. Let i be such that $v \in K_i$. By definition of idle nodes, v satisfies that

$$|N_{G_i}(v)| - d_i(v) < |S_i^v|,$$

where $S_i^v = \{u \in N_{G_i}(v) \cap S_i \ : \ \gamma_i(u) = \gamma_i(v)\}$. It follows that

$$|N_{G_i}(v)| - |S_i^v| < d_i(v) < |L_i(v)|$$

where the second inequality follows from Lemma 9. Since $S_i^v \subseteq N_{G_i}(v)$, we get

$$|N_{G_i}(v) \setminus S_i^v| = |N_{G_i}(v)| - |S_i^v|,$$

and $d_i(v) \leq |L_i(v) \setminus \{\gamma_i(v)\}|$, we get that

$$|N_{G_i}(v) \setminus S_i^v| < |L_i(v) \setminus \{\gamma_i(v)\}|.$$

Therefore, since each time the algorithm removes a color from $L_i(v)$, it also removes a node from $N_{G_i}(v)$, we get that

$$|N_{G_i}(v) \setminus (S_i \cup J_i)| < |L_{i+1}(v)|.$$

For $j = i, \ldots, T$, let us denote by

$$\deg_j^+(v) = N_{G_1}(v) \cap (K_i \cup \cdots \cup K_j \cup V_{j+1})$$

the degree of v in $K_i \cup \cdots \cup K_j \cup V(G_{j+1})$ at the end of iteration j. For $j = i$, we have just shown that $\deg_i^+(v) < |L_{i+1}(v)|$. For $j \geq i$, assuming $\deg_j^+(v) < |L_{j+1}(v)|$, we have

$$\deg_{j+1}^+(v) = \deg_j^+(v) - |N_{G_1}(v) \cap (S_j \cup J_j)|$$

because the nodes in $S_j \cup J_j$ are removed from G_j to get G_{j+1}. Similarly, by the list updates performed by the algorithm, we have

$$L_{j+1}(v) = L_j(v) \setminus \{\delta(u) : u \in N_{G_1}(v) \cap (S_j \cup J_j)\}.$$

Since a node is removed each time a color is removed, we get that $\deg_{j+1}^+(v) < |L_{j+2}(v)|$. Therefore $\deg_j^+(v) < |L_{j+1}(v)|$ holds for every $j \geq i$. The lemma follows from taking $j = T$, because $\deg_T^+(v) = \deg^+(v)$ is the degree of v in $K_i \cup \cdots \cup K_T$ as $V_{T+1} = \varnothing$.

Finally, since, at every iteration i, every active node v truncates its list $L_i(v)$ to be of size exactly $d_i(v) + 1$ at the beginning of the iteration, all lists involved in the algorithm are of size at most $k + 1$. It follows that $\deg^+(v) \leq k$. ◀

The next lemma states that the idle nodes eventually become properly colored.

▶ **Lemma 13.** *For every idle node $v \in K$, and for every $u \in N_{G_1}(v)$, we have $\delta(u) \neq \delta(v)$.*

Proof. Let $v \in K$, and let $u \in N_{G_1}(v)$. If $u \in V \setminus K$, then u terminated at some iteration $i, 1 \leq i \leq T$. By the list update applied at node v, $\delta(u) \notin L_{i+1}(v)$, and thus $\delta(u) \notin L_{T+1}(v)$. By lemma 12, the graph $G_1[K]$ is properly colorable, and is properly colored by the algorithm in Lemma 5. ◀

The next technical lemma concerns the nodes which did not become idle.

▶ **Lemma 14.** *For every $i \geq 1$, and every $v \in S_i \cup J_i$, let*

$$X_i = \{u \in N_{G_i}(v) \cap (S_i \cup J_i) : \delta(u) \neq \delta(v)\}, \text{ and } Y_i = N_{G_i}(v) \setminus (S_i \cup J_i).$$

Then $|X_i| + |Y_i| \geq d_i(v)$.

Proof. Let $i \geq 1$, and $v \in S_i \cup J_i$. From Observation 7, the neighbors of node v can be partitioned into S_i, J_i, and $Y_i = K_i \cup V_{i+1}$. The set $X_i \subseteq S_i \cup J_i$ is the subset of nodes u which adopts their final color as $\delta(u) = \gamma_i(u) \neq \gamma_i(v) = \delta(v)$. The nodes in $(S_i \cup J_i) \setminus X_i$ adopts the same final color as v, i.e., for every $u \in (S_i \cup J_i) \setminus X_i$, we have $\gamma_i(u) = \gamma_i(v)$.

If $v \in S_i$, then the coloring γ_i is such that the neighborhood of v contains all the colors in $L_i(v)$, and thus, thanks to Lemma 8, the neighborhood of v contains at least $d_i(v) + 1$ colors. These colors are those assigned by γ_i to nodes either in Y_i or X_i. Therefore, the neighborhood of v contains at least $d_i(v)$ colors different from $\gamma_i(v)$. Hence the lemma holds for $v \in S_i$.

If $v \in J_i$, then $v \notin K_i$, i.e., v does not become idle at iteration i. As a consequence,

$$|N_{G_i}(v)| - |\{u \in N_{G_i}(v) \cap S_i : \gamma_i = \gamma_i(v)\}| \geq d_i(v).$$

Since $v \in J_i$, we have $N_{G_i}(v) \cap J_i = \varnothing$. Therefore

$$|N_{G_i}(v)| - |\{u \in N_{G_i}(v) \cap (S_i \cup J_i) : \gamma_i = \gamma_i(v)\}| \geq d_i(v).$$

The lemma follows from the fact that the left hand side of the latter inequality is precisely equal to $|X_i| + |Y_i|$. ◀

For computing the total contribution of the neighbors of a node v to the original demand $d(v)$ of that node, we introduce the following notion, which applies to nodes that have not yet terminated, i.e., they are still active, or became idle at previous iterations. For every $i \geq 1$, and $v \in V_i \cup (\cup_{j=1}^{i-1} K_j)$, we denote by

$$\mathsf{good}_i(v) = \{u \in N_{G_1}(v) : \exists j \in \{1, \ldots, i-1\},\ u \in S_j \cup J_j\}$$

the set of neighbors of v in G_1 that have terminated at a round less than i. Note also that, by the list update rules, for every $u \in \mathsf{good}_i(v)$, $\delta(u) \notin L_i(v)$.

▶ **Lemma 15.** *For every $i \geq 1$, and every vertex $v \in V_i \cup (\cup_{j=1}^{i-1} K_j)$, we have*

$$|\mathsf{good}_i(v)| + d_i(v) + |N_{G_1}(v) \cap (\cup_{j=1}^{i-1} K_j)| \geq d(v).$$

Proof. The proof is by induction on $i \geq 1$. For $i = 1$, $\mathsf{good}_1(v) = \varnothing$, $d_1(v) = d(v)$, and $\cup_{j=1}^{0} K_j = \varnothing$, so the lemma holds. Let us now assume that this lemma is true for $i \geq 1$. By the definition of good vertices, $\mathsf{good}_{i+1}(v) = \mathsf{good}_i(v) \cup (N_{G_i}(v) \cap (S_i \cup J_i))$. Since, $\mathsf{good}_i(v) \cap (N_{G_i}(v) \cap (S_i \cup J_i)) = \varnothing$, it follows that

$$|\mathsf{good}_{i+1}(v)| = |\mathsf{good}_i(v)| + |(N_{G_i}(v) \cap (S_i \cup J_i))|. \tag{2}$$

Now, by the updates of the demands performed in the algorithm,

$$d_{i+1}(v) = d_i(v) - |N_{G_i}(v) \cap (S_i \cup J_i \cup K_i)|.$$

Since S_i, J_i, and K_i are disjoint, we can rewrite the previous equation as

$$d_{i+1}(v) = d_i(v) - (|N_{G_i}(v) \cap S_i| + |N_{G_i}(v) \cap J_i| + |N_{G_i}(v) \cap K_i|). \tag{3}$$

Finally, we have that

$$|N_{G_1}(v) \cap (\cup_{j=1}^{i} K_j)| = \sum_{j=1}^{i} |N_{G_1}(v) \cap K_j|. \tag{4}$$

Adding Equations (2), (3), and (4), and using the fact that S_i, J_i, and K_i are disjoint, we get

$$|\mathsf{good}_{i+1}(v)| + d_{i+1}(v) + |N_{G_1}(v) \cap (\cup_{j=1}^{i} K_j)| = |\mathsf{good}_i(v)| + d_i(v) + \sum_{j=1}^{i} |N_{G_1}(v) \cap K_j| - |N_{G_i} \cap K_i|.$$

Since $N_{G_1}(v) \cap K_i = N_{G_i} \cap K_i$ for every $i \geq 1$, we obtain that

$$\begin{aligned} |\mathsf{good}_{i+1}(v)| + d_{i+1}(v) + |N_{G_1}(v) \cap (\cup_{j=1}^{i} K_j)| &= |\mathsf{good}_i(v)| + d_i(v) + \sum_{j=1}^{i-1} |N_{G_1}(v) \cap K_j| \\ &= |\mathsf{good}_i(v)| + d_i(v) + |N_{G_1}(v) \cap (\cup_{j=1}^{i-1} K_j)|. \end{aligned}$$

The claim then follows from the induction hypothesis. ◀

We can now conclude with the correctness of our algorithm.

▶ **Proposition 16.** *The algorithm described in Section 2 terminates, and the coloring δ returned by the algorithm is a solution to k-partial list-coloring in G.*

Proof. The fact that the algorithm execute a finite number of iterations has been established in Lemma 11. For every vertex $v \in V$, either v terminates at some iteration $i \in \{1, \ldots, T\}$, or v becomes idle at some iteration $i \in \{1, \ldots, T\}$.

Let us consider a node v that terminates at some iteration $i \in \{1, \ldots, T\}$. By Lemma 14, and the fact that if $u \in N_{G_i}(v) \setminus (S_i \cup J_i)$, then $\delta(v) \notin L_i(u)$, we deduce that the number of neighbors of v in G_i which will take different color that v is at least $d_i(v)$. As v is active in the beginning of round i, it follows from Lemma 15 that

$$|\mathsf{good}_i(v)| + d_i(v) + |N_{G_1}(v) \cap K| \geq d(v).$$

Thanks to the definition of good vertices, Lemma 13, and the fact that $d(v) \leq |N_{G_1}(v)|$, it results that the lemma holds for terminating nodes.

Let us now consider a node that becomes idle at some iteration $i \in \{1, \ldots, T\}$. In that case, the results immediately follows from Lemma 13. ◀

3.2 Complexity Analysis

We now prove that our algorithm terminates in the prescribed number of rounds.

▶ **Proposition 17.** *The algorithm described in Section 2 terminates in $O(\log n \cdot \log^3 k)$ rounds.*

Proof. The algorithm starts with a preprocessing stage which takes $O(\log^\star n)$ rounds [11]. The removal of the monochromatic edges from the graph takes $O(1)$ rounds. For each iteration i of the algorithm, the prepocessing phase takes $O(1)$ rounds for each vertex to select incident edges and remove unselected edges. The derandomization phase takes $O(\log^2 k)$ rounds by Lemma 3. Finally, for the color assignment phase, computing the graph G'_i takes $O(1)$ rounds, and computing the MIS I_i on H_i takes $O(\log^\star k)$ rounds [12]. Therefore, each iteration takes at most $O(\log^2 k)$ rounds. The assignment of colors to the terminated vertices, and the update of the graph from G_i to G_{i+1} takes $O(1)$ rounds.

Thanks to Lemma 10, the graph H_i induced by the nodes with degree less than 16 in $G'_i = (V_i, F_i)$ has at least $\frac{|V_i|}{2}$ vertices. For every maximal independent set I_i in H_i, every $v \in I_i$ dominates at most 16 nodes (itself, plus its at most 15 neighbors). Therefore, at least

$\frac{|V_i|}{32}$ vertices are in I_i during the ith iteration. Each vertex in I_i either terminate (in S_i or J_i) or become idle (in K_i). Therefore, at most $\frac{31|V_i|}{32}$ vertices participate in the next iteration. Therefore, the number of iterations for the core of the algorithm is at most $O(\log n)$. It follows that, in total, the core of the algorithm takes at most $O(\log^2 k \cdot \log n)$ rounds.

For the post processing phase, by lemma 12, $G_1[K]$ is a layered graph for which $\hat{\Delta} = \max_{v \in K} \deg^+(v) \leq k$. By lemma 5, $G_1[K]$ it takes at most $O(\log^3 k \cdot \log n)$ rounds to properly list-color the vertices of this graph. ◀

Theorem 1 directly follows from Propositions 16 and 17.

4 Conclusion

We have shown that the breakthrough result of Ghaffari and Kuhn [10], stating that $(\Delta+1)$-list-coloring can be computed in $O(\log n \cdot \log^2 \Delta)$ rounds, can be generalized to k-partial list coloring, where k is the maximum demand, for all $k \geq 0$. Our algorithm for k-partial list-coloring runs in $O(\log n \cdot \log^3 k)$ rounds. The extra $\log k$ factor is due to the post-processing stage, for assigning colors to the idle nodes, which itself comes from the extra $\log k$ factor in proper list-coloring of layered graphs in [10]. However, in the specific case of k-partial $(k+1)$-coloring, this extra $\log k$ factor can be avoided, and our algorithm runs in $O(\log n \cdot \log^2 k)$ rounds. It would be interesting to know whether this extra $\log k$ factor could be avoided for general partial list-coloring, and of course whether $O(\log n \cdot \log^2 k)$ rounds is the best that can be achieved for k-partial $(k+1)$-coloring for general k.

Another interesting generalization of proper coloring is to consider the extension of proper Δ-coloring to k-partial k-coloring, for $k = 1, \ldots, \Delta$. Brook's theorem states that, in a connected graph with maximum degree Δ, the vertices can be properly colored with only Δ colors in all graphs, except for complete graphs, and cycle graphs of odd length, which require $\Delta+1$ colors. The algorithm in [10] can be used to properly Δ-color every Δ-colorable graph in $O(\log^2 n \log^2 \Delta)$ rounds. We do not know whether this can be generalized to k-partial k-coloring every graph for which there exist a k-partial coloring using a palette with only k colors.

References

1 Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–369, 1989. doi:10.1109/SFCS.1989.63504.

2 Alkida Balliu, Juho Hirvonen, Christoph Lenzen, Dennis Olivetti, and Jukka Suomela. Locality of not-so-weak coloring. In *26th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 11639 of *LNCS*, pages 37–51. Springer, 2019. doi:10.1007/978-3-030-24922-9_3.

3 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

4 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 *37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 437–446, 2018. URL: https://dl.acm.org/citation.cfm?id=3212769.

5 Sebastian Brandt. An automatic speedup theorem for distributed problems. In *38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 379–388, 2019. doi:10.1145/3293611.3331611.

6 Salwa Faour, Mohsen Ghaffari, Christoph Grunau, Fabian Kuhn, and Václav Rozhon. Local distributed rounding: Generalized to mis, matching, set cover, and beyond. In *34th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4409–4447, 2023. doi:10.1137/1.9781611977554.CH168.

7 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In Irit Dinur, editor, *57th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 625–634, 2016. doi:10.1109/FOCS.2016.73.

8 Mohsen Ghaffari and Christoph Grunau. Faster deterministic distributed MIS and approximate matching. In *55th ACM Symposium on Theory of Computing (STOC)*, pages 1777–1790, 2023. doi:10.1145/3564246.3585243.

9 Mohsen Ghaffari, Christoph Grunau, Bernhard Haeupler, Saeed Ilchi, and Václav Rozhon. Improved distributed network decomposition, hitting sets, and spanners, via derandomization. In *34th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2532–2566, 2023. doi:10.1137/1.9781611977554.CH97.

10 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.

11 Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 138–144, 2009. doi:10.1145/1583991.1584032.

12 Nathan Linial. Distributive graph algorithms-global solutions from local data. In *28th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 331–335, 1987. doi:10.1109/SFCS.1987.20.

13 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

14 Yannic Maus and Tigran Tonoyan. Local conflict coloring revisited: Linial for lists. In *34th International Symposium on Distributed Computing (DISC)*, volume 179 of *LIPIcs*, pages 16:1–16:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.DISC.2020.16.

15 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

16 Alessandro Panconesi and Aravind Srinivasan. On the complexity of distributed network decomposition. *J. Algorithms*, 20(2):356–374, 1996. doi:10.1006/JAGM.1996.0017.

17 David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.

18 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *52nd ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020. doi:10.1145/3357713.3384298.

A Partial List-Coloring with a Common Color

In this section, we consider k-partial list-coloring with the additional assumption that all lists share a common "special" color c^* that is known to all nodes. This setting includes the important case of k-partial $(k+1)$-coloring, with $c^* = 1$. In this context, it is easier to assume that each node v has a specific list of colors of size at least $d(v)$ not containing c^*, and the color outputted by each node must be either c^* or a color from its list $L(v)$.

A.1 The Algorithm

Our algorithm remains based on the structure of the algorithm in [10] for $(\Delta+1)$-list-coloring, as our algorithm for general k-partial list-coloring, but the color assignment phase is much easier, thanks to the presence of the special color c^*, which can be used as a backup for saturated nodes. Specifically, our algorithm works as follows.

- **Preprocessing Stage.** This stage is identical to what was done in the algorithm for general k-partial list-coloring. Specifically, we compute a k-partial $O(k^2)$-coloring of G, which we denote by α. Again, this can be done in $O(\log^* n)$ rounds [11]. Remove the monochromatic edges to obtain a sub-graph of G, denoted by $G_1 = (V_1, E_1)$. Note that in G_1 the degree of any node v is at least $\min\{k, \deg(v)\}$, and therefore at least $d(v)$. We further arbitrarily truncate each list $L(v)$ to be of size $d(v)$.
- **Computing the coloring δ.** Initially, $\delta(v) = \perp$ for every $v \in V$. For computing their colors, the nodes performs $O(\log n)$ iterations. At each iteration, some nodes $v \in V$ fix their colors $\delta(v) \in L(v) \cup \{c^*\}$. Once a node becomes colored (i.e., different from $\perp$), its color never changes. We proceed with another iteration as long as there is at least one node v with $\delta(v) = \perp$. Note that, as opposed to the previous algorithm for general k-partial list-coloring, there is no post-processing stage.

  For every $v \in V$, we set $d_1(v)$ to be the input demand $d(v)$, and we set $L_1(v)$ to be the input list $L(v)$. The input to iteration $i \geq 1$ is a graph $G_i = (V_i, E_i)$, a demand function $d_i : V_i \to \mathbb{N}$, and a color list function $L_i : V_i \to 2^{\mathbb{N}}$.

  We now describe one iteration of the algorithm.

  1. **Preprocessing Phase.** Every vertex v arbitrarily selects $d_i(v)$ many edges incident on it, and orient them as outgoing edges, as we did before in the general k-partial list-coloring algorithm. Note that an edge can be oriented both ways, by its two endpoints. Any unoriented edge is removed from G_i. In what follow, we slightly abuse notations, and $G_i = (V_i, E_i)$ now refers to the graph after the removal of these edges. We define

     $$N^{out}_{G_i}(v) \coloneqq \{u \in N_{G_i}(v) \ : \ v \text{ selected } e = \{u, v\} \text{ as outgoing edge}\}.$$

  2. **Derandomization Phase.** This phase is identical to the derandomization phase of the general k-partial list-coloring, excepted that we are using lists of $d(v)$ colors at nodes v, instead of lists of $d(v) + 1$ colors, as nodes can use the wildcard color c^* in addition to the colors in their list. Corollary 3.6 in [10] provides a way to "derandomize" the uniform distribution while almost preserving the weights assigned by w_p (cf. Lemma 3).
  3. **Color Assignment Phase.** Given the graph $G_i = (V_i, E_i)$, and the coloring γ from Lemma 3, we consider the graph $G'_i = (V_i, F_i)$ where

     $$F_i \coloneqq \big\{\{u, v\} \in E_i \ : \ \gamma(u) = \gamma(v)\big\}.$$

     As we have seen in Lemma 10, we have $|F_i| \leq 4 \cdot |V_i|$. As a consequence, the number of nodes $v \in V_i$ with degree $\deg_{G'_i}(v) \geq 16$ is at most $|V_i|/2$. We denote by H_i the subgraph of G'_i induced by the nodes with $\deg_{G'_i}(v) < 16$. By construction, H_i has maximum degree at most $15 = O(1)$. In addition, H_i has a proper $(O(k^2))$ coloring (by the preprocessing stage operations). It follows that the nodes of H_i can compute a maximal independent set (MIS) I_i in H_i in $O(\log^* k)$ rounds [12] . Note that, since each of the at least $|V_i|/2$ nodes in H_i has maximum degree 15, we have $|I_i| \geq \frac{|V_i|}{32}$.

     We now modify the notion of *saturated* nodes, for adapting it to the setting in which all lists share a common color c^*.

     ▶ **Definition 18.** *We say that a node $v \in V_i$ is saturated by I_i with respect to the coloring γ if the following condition holds: for every $c \in L_i(v)$, there exists a neighbor $u \in N_{G_i}(v) \cap I_i$ with $\gamma(u) = c$.*

     Let us denote by S_i the set of nodes saturated by I_i w.r.t. γ. It is easy to see that $S_i \cap I_i = \varnothing$ (cf. Lemma 22).

All nodes in $I_i \cup S_i$ fix their final δ-colors in a way different from the color assignment in the case of general k-partial list-coloring, using the existence of the common "wildcard" color c^*, as follows:

- **MIS Node Rule:** if $v \in I_i$, then $\delta(v) \leftarrow \gamma(v)$, i.e., v adopts its γ-color as its final color, and
- **Saturated Node Rule:** if $v \in S_i$, then $\delta(v) \leftarrow c^*$, i.e., v adopts the wildcard color c^* as its final color.

Towards the next iteration we perform the following updates.

- The list of colors available to each node $v \in V_i \setminus (I_i \cup S_i)$ is updated as follows.

$$L_{i+1}(v) \leftarrow L_i(v) \setminus \{\delta(u) : \ u \in N_{G_i}(v) \cap I_i\} \ .$$

- The graph itself is updated to be $G_{i+1} = (V_{i+1}, E_{i+1})$ as
  - **a.** $V_{i+1} = V_i \setminus (S_i \cup I_i)$
  - **b.** $E_{i+1} = \big\{\{u, v\} \in E_i : u \notin S_i \cup I_i \text{ and } v \notin S_i \cup I_i\big\}$

- The (remaining) demands for each node $v \in V_i \setminus (I_i \cup S_i)$ are defined as follows.
  - **a.** If $\deg_{G_{i+1}}(v) < |L_{i+1}(v)|$ then

$$d_{i+1}(v) \leftarrow \max\{0, d_i(v) - |N_{G_i}(v) \cap I_i| - |N_{G_i}(v) \cap S_i|\};$$

  - **b.** Otherwise

$$d_{i+1}(v) \leftarrow \max\{0, d_i(v) - |N_{G_i}(v) \cap I_i|\}.$$

This completes the description of our algorithm.

A.2 The Proof

We first prove that our algorithm does solve k-partial list coloring under the assumption of the existence of the special color c^*. Next, we will prove that it runs in the prescribed number of rounds. Thanks to Lemma 6, we merely prove the correctness of the algorithm on the graph G after the preprocessing stage. We start by a claim on the relation between the size of the list of each node and its demand over the iterations.

▶ **Lemma 19.** *For any $j \geq i$, if $d_j(v) > 0$ then $|L_j(v)| - d_j(v) \geq |L_i(v)| - d_i(v)$.*

Proof. We prove the claim by induction on $j \geq i$. The base case being $j = i$ is obviously true. Assume the induction hypothesis for $j \geq i$. According to the algorithm

$$L_{j+1}(v) \coloneqq L_j(v) \setminus \{\delta(u) : \ u \in N_{G_j}(v) \cap I_j\}.$$

Now, according to the algorithm, if $d_{j+1}(v) > 0$ then

$$d_{j+1}(v) \leq d_j(v) - |N_{G_j}(v) \cap I_j|.$$

The induction claim follows from

$$|\{\delta(u) : \ u \in N_{G_j}(v) \cap I_j\}| \leq |N_{G_j}(v) \cap I_j|,$$

and from the induction hypothesis. ◀

▶ **Lemma 20.** *For every iteration i and every node $v \in V_i$, $|L_i(v)| \geq d_i(v)$.*

Proof. The claim follows from the definition of the input to the algorithm and from Lemma 19. ◀

▶ **Lemma 21.** *For every iteration i and every node $v \in V_i$, $\deg_{G_i}(v) \geq d_i(v)$.*

Proof. In each iteration i of the algorithm the degree of a node v is reduced by exactly

$$|N_{G_i}(v) \cap I_i| + |N_{G_i}(v) \cap S_i|,$$

and its demand is reduced by at most that number. ◀

The following establishes that our algorithm is well defined.

▶ **Lemma 22.** *For every iteration i, we have $S_i \cap I_i = \varnothing$.*

Proof. if $v \in S_i$, then v has a neighbor $u \in N^{\gamma}_{G_i}(v) \cap I_i$ with $\gamma(u) = \gamma(v)$. As a consequence, if $v \in I_i$ too, then both v and u were vertices of H_γ that were adjacent in this graph, a contradiction with the fact that I_i is an independent set of H_γ. ◀

We now give two lemmas that will allow us to establish the number of bichromatic edges in G_i, according to the final coloring δ.

▶ **Lemma 23.** *For every iteration i, every node $u \in I_i$, and every node $v \in N_{G_i}(u)$, we have*

$$\begin{cases} \delta(u) \neq \delta(v) & \text{if } v \in I_i \cup S_i \\ \delta(u) \notin L_{i+1}(v) & \text{otherwise} \end{cases}$$

Proof. First note that it can be the case that u and v are both in I_i and adjacent in G_i, because I_i is an independent set in H_i and not in G_i.

- If $v \in I_i$, then $\delta(u) = \gamma(u) \neq \gamma(v) = \delta(v)$, as otherwise the edges (u, v) would have been in H_i (because H_i is obtained from G_i by removing bichromatic edges and nodes), and therefore it cannot be the case that both u and v are in I_i.
- If $v \in S_i$ then $\delta(v) = c^* \notin L(u)$ and therefore $\delta(u) \neq \delta(v)$. ◀

▶ **Lemma 24.** *For any iteration i and any node $u \in S_i$,*

$$|\{v : v \in N_{G_i}(u), \delta(u) \neq \delta(v)\}| \geq d_i(u) \ .$$

Proof. By the definition of S_i it follows that for every $c \in L_i(u)$, there exists a neighbor $v \in N_{G_i}(v) \cap I_i$ with $\gamma(v) = c$. By the algorithm for all nodes $x \in I_i$, $\delta(x) = \gamma(x) \neq c^*$, and $\delta(u) = c^*$. By Lemma 20, $|L_i(u)| \geq d_i(u)$, and the lemma follows. ◀

We proceed with two definitions. First, we define the notion of *slackness*.

▶ **Definition 25.** *Let $i \geq 1$, and let $v \in V_i$ be a vertex. The* slackness *of v is defined as* $\texttt{slack}_i(v) := \deg_{G_i}(v) - d_i(v)$.

Second, we introduce the notion of *free* nodes.

▶ **Definition 26.** *Let $i \geq 1$, and let $v \in V_i$ be a vertex. We say that v is* free *whenever $\deg_i(v) < |L_i(v)|$. We denote by free_i the set of free nodes in G_i.*

▶ **Lemma 27.** *For $v \in V_i$, if $|N_{G_i}(v) \cap S_i| > \texttt{slack}_i(v)$ then, for any $j > i$,*

$$v \in V_j \Rightarrow v \in \mathsf{free}_j.$$

Proof. By the definition of the algorithm

$$\deg_{i+1}(v) = \deg_i(v) - |N_{G_i}(v) \cap I_i| - |N_{G_i}(v) \cap S_i|.$$

Using the conditions of the Lemma we then have that

$$\begin{aligned}\deg_{i+1}(v) &< \deg_i(v) - |N_{G_i}(v) \cap I_i| - \mathtt{slack}_i(v) \\ &= \deg_i(v) - |N_{G_i}(v) \cap I_i| - (\deg_i(v) - d_i(v)) \\ &= d_i(v) - |N_{G_i}(v) \cap I_i|.\end{aligned}$$

Using Lemma 20 we have then

$$\deg_{i+1}(v) < |L_i(v)| - |N_{G_i}(v) \cap I_i| \leq |L_{i+1}(v)|.$$

We therefore have that $\deg_{i+1}(v) < |L_{i+1}(v)|$. By Lemma 19 we have that, for all $i < j$, $\deg_j(v) < |L_j(v)|$. By the definition of a free node, for all $i < j$, if $v \in V_j$ then $v \in \mathsf{free}_j$. ◀

We need one last definition, which adapts the notion of good nodes to the case of lists with a common color. The new notion of *good nodes for* v (at iteration i) is intended to allow us to count the number of nodes adjacent to v in G that are already certain to be colored differently than v by the final coloring.

▶ **Definition 28.** *For iteration i and node $v \in V_i$ we define* $\mathsf{good}_i(v)$ *as follows.*

$$\mathsf{good}_i(v) = \begin{cases} \{u \in V \setminus V_i \mid \delta(u) \notin L_i(v)\} & \text{if } v \in \mathsf{free}_i \\ \{u \in V \setminus V_i \mid \delta(u) \notin L_i(v) \cup \{c^*\}\} & \text{otherwise.} \end{cases}$$

The following observation directly follows from the definition of $\mathsf{good}_i(v)$. It formally states which of the neighbors of v in G_{i-1} are added, at the end of iteration $i-1$, to the set of "good neighbors" of v.

▶ **Observation 29.** *For $i > 1$, if $v \in \mathsf{free}_i$, then*

$$N_{G_{i-1}}(v) \cap (S_{i-1} \cup I_{i-1}) \subseteq \mathsf{good}_i(v),$$

else

$$N_{G_{i-1}}(v) \cap I_{i-1} \subseteq \mathsf{good}_i(v).$$

▶ **Lemma 30.** *For every iteration i and every node $v \in V$*

$$|\mathsf{good}_i(v)| + d_i(v) \geq d(v)\ .$$

Proof. We prove the claim by induction on i. The base case being $i = 1$ holds since $d_1(v) = d(v)$ by the definition of the algorithm. Assume the induction hypothesis for $i \geq 1$. First observe that for all $v \in V$, $\mathsf{good}_i(v) \subseteq \mathsf{good}_{i+1}(v)$ (and hence $|\mathsf{good}_i(v)| \leq |\mathsf{good}_{i+1}(v)|$), because

$$V \setminus V_i \subseteq V \setminus V_{i+1},\ \ L_{i+1} \subseteq L_i, \text{ and } \mathsf{free}_i \subseteq \mathsf{free}_{i+1}.$$

However, it could be that $d_{i+1}(v) < d_i(v)$. According to the algorithm there are two cases for the update of $d_{i+1}(v)$.

– If $\deg_{G_{i+1}}(v) < L_{i+1}(v)$ then $d_{i+1}(v)$ is reduced, compared to $d_i(v)$, by at most

$$|N_{G_i}(v) \cap I_i| + |N_{G_i}(v) \cap S_i|.$$

Moreover, if $\deg_{G_{i+1}}(v) < L_{i+1}(v)$ then $v \in \mathsf{free}_{i+1}$, and by Observation 29,

$$|\mathsf{good}_{i+1}(v) \setminus \mathsf{good}_i(v)| \geq |N_{G_i}(v) \cap I_i| + |N_{G_i}(v) \cap S_i|.$$

– If $\deg_{G_{i+1}}(v) \geq L_{i+1}(v)$ then $d_{i+1}(v)$ is reduced, compared to $d_i(v)$, by at most $|N_{G_i}(v) \cap I_i|$. Using again Observation 29,

$$|\mathsf{good}_{i+1}(v) \setminus \mathsf{good}_i(v)| \geq |N_{G_i}(v) \cap I_i|,$$

which completes the proof. ◀

We now have all the necessary ingredients to establish the correctness of our algorithm.

▶ **Proposition 31.** *If the algorithm in Section A.1 terminates, then the final coloring δ is a solution to k-partial list-coloring on G, whenever all lists include a common color c^*.*

Proof. Let vertex v terminate at the end of iteration i of the algorithm (i.e., $v \in V_i$ but $v \notin V_{i+1}$).

- If $v \in I_i$, then, by Lemma 23, for every $u \in N_{G_i}(v)$, either $\delta(u) \neq \delta(v)$ or $\delta(u) \notin L_{i+1}(v)$. Now, observe that $\mathsf{good}_i(v) \cap V_i = \varnothing$. Hence, the number of nodes $u \notin \mathsf{good}_i(v), u \in_{G_i} (v)$ colored differently than v is at least $|N_{G_i}(v)| \geq d_i(v)$, using Lemma 21.
- If $v \in S_i$, then, by Lemma 24,

  $$|\{u \in N_{G_i}(v) \ : \ \delta(u) \neq \delta(v)\}| \geq d_i(v).$$

  Similarly to the previous case, because $\mathsf{good}_i(v) \cap V_i = \varnothing$, the number of nodes $u \notin \mathsf{good}_i(v), u \in_{G_i} (v)$ is at least $d_i(v)$.

By Lemma 30 we get that the total number of nodes $u \in N_G(v)$ colored differently than v is at least $d(v)$. ◀

We finally prove that our algorithm terminates in the prescribed number of rounds.

▶ **Proposition 32.** *The algorithm in Section A.1 terminates in $O(\log n \cdot \log^2 k)$ rounds.*

Proof. By the same arguments as in the proof of Proposition 17, excepted from the post-processing stage, which is absent from the algorithm for lists with a common color. ◀

Theorem 2 directly follows from Propositions 31 and 32.

A Tight Bound on Multiple Spending in Decentralized Cryptocurrencies

João Paulo Bezerra
Télécom Paris, Institut Polytechnique de Paris, France

Petr Kuznetsov
Télécom Paris, Institut Polytechnique de Paris, France

Abstract

The last decade has seen a variety of Asset-Transfer systems designed for decentralized environments. The major problem these systems address is *double-spending*, and solving it inherently imposes strong *trust* assumptions on the system participants. In this paper, we take a non-orthodox approach to the double-spending problem that might suit better realistic environments in which these systems are to be deployed. We consider the *decentralized trust* setting, where each user may independently choose who to trust by forming their local quorums. In this setting, we define *k-Spending Asset Transfer*, a relaxed version of asset transfer which bounds the number of times a system participant may spend an asset it received. We establish a precise relationship between the decentralized trust assumptions and k, the optimal *spending number* of the system.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Quorum systems, decentralized trust, consistency measure, asset transfer, accountability

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.31

Related Version The full version of the paper is available as a technical report.
Full Version: `https://arxiv.org/abs/2205.14076`

Acknowledgements This work was supported by TrustShare Innovation Chair.

1 Introduction

Fault models and quorum systems. Distributed protocols, such as consensus and broadcast, are generally built to be resilient against arbitrary (Byzantine) faults of system members. To maintain consistency and progress, these protocols typically have to assume that only a certain fraction of system members are allowed to be Byzantine. In the special case of a *uniform* fault model, where faults of system members are identically and independently distributed, bounds on the number f of Byzantine members that can be tolerated are well known: less than half of system members ($f < n/2$) in synchronous networks (using digital signatures) [27], and less than one third ($f < n/3$) in asynchronous or partially synchronous networks [7].

More general fault models can be captured via *quorum systems* [33, 40], collections of subsets of system participants, called *quorums*, that meet two conditions: in every system run, *every* two quorums should have at least one correct participant in common and *some* quorum should only contain correct participants. Intuitively, quorums encapsulate *trust* the system members express to each other. Every quorum can act on behalf of the whole system: an update or a query on the data is considered safe if it involves a trusted set of replicas.

Decentralized quorums. Conventionally, quorum assumptions are centralized: all participants share the same quorum system. In some large-scale distributed systems, it might be, however, difficult to expect that all participants come to the same trust assumptions.

Recently, the quorum-based approach to system design has been explored in a completely new way. It started with system implementations [37, 34] that allowed their users to not necessarily hold the same assumptions of who to trust, i.e., to maintain *local* quorum systems. Based on its local knowledge, a system member might have its own idea about which subsets of other participants are trustworthy and which are not. We come, therefore, to the model of *decentralized quorums*: each system member maintains its own quorum system.

Great effort has been invested into improving protocols designed for uniform fault models [12, 28, 21, 42], or in understanding which conditions on individual quorum systems are necessary and sufficient, so that some well-defined subset of participants can solve a problem [19, 8, 32]. However, little is understood about the "damage" that Byzantine processes might cause if these conditions do not hold, e.g., in the decentralized quorum system model. Intuitively, the more Byzantine processes there are or more strategically they are located in decentralized quorums, the more important is the impact they have on the system's consistency. But what exactly does "more important" mean here?

Asynchronous cryptocurrencies. In this paper, we study this question on the example of asset-transfer systems (or *cryptocurrencies*). Conventionally, the major challenge addressed by a cryptocurrency is to prevent *double spending*, when a malicious or misconfigured user manages to spend the same coin more than once. As was originally claimed by Nakamoto [36], preventing double spending in systems with mutual distrust requires honest users to agree on the order in which the transactions must be executed, i.e., to solve the fundamental problem of *consensus* [18]. Bitcoin achieves probabilistic *permissionless* consensus assuming a synchronous system and using the proof-of-work mechanism. The protocol is notoriously energy-consuming and slow. Since then, a long line of systems used consensus for implementing cryptocurrencies in both permissionless and permissioned contexts.

It has been later observed that cryptocurrencies do not always require consensus in general [23, 22]. It turns out that it is not always necessary to maintain a totally ordered set of transactions, a specific partial order may suffice. Intuitively, if we assume that each account has a single dedicated owner, it is sufficient to agree on the order of outgoing transactions *per account*. Transactions operating on different accounts can be ordered arbitrarily without affecting correctness. Double spending is excluded, as no user can publish "conflicting" transactions on its account (spending more money than its account holds). Recently proposed asynchronous (*consensus-free*) cryptocurrencies [4, 15] exhibit significant advantages over consensus-based protocols in terms of scalability, performance and robustness. However, as they still rely on classical quorum systems, they are challenging to apply at a large scale.

Contributions. In this paper, we explore the potential of *decentralized* quorums in implementing asynchronous cryptocurrencies. Naturally, this model allows us to formally capture the *double spending* phenomenon. In a way, we mimic the principle followed by real-world financial systems, where double spending is a routine phenomenon.

We introduce *k-Spending Asset Transfer*, a relaxed cryptocurrency abstraction suitable for decentralized trust models. Notice that in this model, quorums chosen by correct processes might not be *globally consistent*, i.e., some quorums might not overlap on a correct process. Byzantine processes can exploit this lack of consistency by enforcing correct processes to accept conflicting transactions with the same input, resulting in *multiple spending*.

Intuitively, a k-spending asset-transfer system guarantees that once a participant receives an asset, it spend it at most k times. It ensures, however, that any instance of multiple-spending that affects correct participants should be eventually detected and a proof of misbehaviour against the Byzantine process should be published.

As a bold analogy, one can think of a global financial trading system, where every national economy benefits from mutual trust, while cross-border interactions are less reliable. But if the lack of trust is exploited by a cheating trader, correct participants should eventually be able to detect and punish the cheater by, e.g., excluding it from the system.

We show how the parameter k in k-spending asset transfer relates to the structure of the underlying quorum assumptions. We visualize these assumptions via a family $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$ of graphs, one for each possible faulty set $F \in \mathcal{F}$ and each quorum map S, mapping each process p to an element in its local quorum system $\mathcal{Q}(p)$. It turns out that the optimal number of times a coin can be spent in this system is precisely the maximum *independence number* over graphs in $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$.

Thus, our contributions are three-fold. We introduce the abstraction of k-spending asset transfer that defines a precise bound k on the number of times a given asset can be spent, once it is received by a system participant. We represent decentralized trust assumption in the form of a family of trust graphs and show that its maximum independence number gives a lower bound on k. We present a k-asset transfer implementation that shows that the bound is tight. In addition, the algorithm maintains an accountability mechanism that keeps track of multiple spending and publishes evidences of misbehavior.

Road map. The rest of the paper is organized as follows. In Section 2 we present our system model. Section 3 introduces a graph representation of trust, used later in the paper to prove lower bounds on "the amount of inconsistency" in cryptocurrency implementations. In Section 4 we give the specification of *k-Spending Asset Transfer* (k-AT) and present a protocol for implementing it. We show that our k-AT algorithm is optimal in Section 5, by relating it to a relaxed broadcast abstraction: *k-Consistent Broadcast* (k-CB). We overview related work in Section 6. Finally, we discuss the results and future work in Section 7.

2 System Model

Processes. A system is composed of a set of *processes* $\Pi = \{p_1, ..., p_n\}$. Every process is assigned an *algorithm* (we also say *protocol*), an automaton defined as a set of possible *states* (including the *initial state*), a set of *events* it can produce and a transition function that maps each state to a corresponding new state. An event is either an *input* (a call operation from the application or a message received from another process) or an *output* (a response to an application call or a message sent to another process); *send* and *receive* denote events involving communication between processes.

Executions and failures. A *configuration* C is a collection of states of all processes. In addition, C^0 is used to denote a special configuration where processes are in their initial states. An *execution* (or a *run*) Σ is a sequence of events, where every event is associated with a distinct process and every *receive*(m) event has a preceding matching *send*(m) event. A process *misbehaves* in a run (we also call it *Byzantine*) if it produces an event that is not prescribed by the assigned protocol, given the preceding sequence of events, starting from the initial configuration C^0. If a process does not misbehave, we call it *benign*. In an infinite run, a process *crashes* if it prematurely stops producing events required by the protocol; if a process is benign and never crashes we call it *correct*, and it is *faulty* otherwise. Let *part*(Σ) denote the set of processes that produce events in an execution Σ.

Channels. Every pair of processes communicate over a *reliable channel*: in every infinite run, if a correct process p sends a message m to a correct process q, m eventually arrives, and q receives a message from p only if p sent it. We impose no synchrony assumptions. In particular, we assume no bounds on the time required to convey a message from one correct process to another.

Digital signatures. We use asymmetric cryptographic tools: a pair public-key/private-key is associated with every process in Π [9]. The private key remains secret to its owner and can be used to produce a *signature* for a statement, while the public key is known by all processes and is used to *verify* that a signature is valid. Every process have access to operations *sign* and *verify*: *sign* takes the process' identifier and a bit string as parameters and returns a signature, while *verify* takes the process' identifier, a bit string and a signature as parameters and return $b \in \{TRUE, FALSE\}$. We assume a computationally bound adversary: no process can forge the signature for a statement of a benign process.

Trust assumptions. We now define our decentralized trust model. A *quorum system map* $\mathcal{Q} : \Pi \to 2^{2^\Pi}$ provides every process with a set of process subsets: for every process p, $\mathcal{Q}(p)$ is the set of *quorums of* p. We assume that p includes itself in each of its quorums: $\forall Q \in \mathcal{Q}(p) : p \in Q$. Intuitively, $\mathcal{Q}(p)$ consists of sets of processes p expects to appear correct in system runs. From p's perspective, for every quorum $Q \in \mathcal{Q}(p)$, there must be an execution in which Q is precisely the set of correct processes. However, these expectations may be violated by the environment. We therefore introduce a *fault model* $\mathcal{F} \subseteq 2^\Pi$ (sometimes also called an *adversary structure*) stipulating which process subsets can be faulty. We assume *inclusion-closed* fault models that, intuitively, do not force processes to fail: $\forall F \in \mathcal{F},\ F' \subseteq F :\ F' \in \mathcal{F}$. From now on, we consider only executions Σ that *complies with* $\mathcal{F}$, i.e., the set of faulty processes in Σ is in $\mathcal{F}$.

Given a faulty set $F \in \mathcal{F}$, a process p is called *live under* F if it has a *live quorum*, i.e., $\exists Q \in \mathcal{Q}(p) : Q \cap F = \emptyset$. For example, let the uniform f*-resilient* fault model: $\mathcal{F} = \{F \subseteq \Pi : |F| \leq f\}$. If $\mathcal{Q}(p)$ includes all sets of $n - f$ processes, then p is guaranteed to have at least one live quorum in every execution. On the other hand, if $\mathcal{Q}(p)$ expects that a selected process $q \neq p$ is always correct ($q \in \cap_{Q \in \mathcal{Q}(p)} Q$), then p is not live in any execution with a faulty set such that $q \in F$.

In the rest of the paper, we consider a *trust model* $(\mathcal{Q}, \mathcal{F})$, where $\mathcal{Q}$ is a quorum map and $\mathcal{F}$ is a fault model.

3 Graph Representation of Trust

We use undirected graphs to depict possible scenarios of executions with trust assumptions $(\mathcal{Q}, \mathcal{F})$. Intuitively, each graph represents a situation where a correct process hears from a quorum before accepting a statement in a protocol. Let $S : \Pi \to 2^\Pi, S(p) \in \mathcal{Q}(p)$, be a map providing each process with one of its quorums, and $\mathcal{S}$ be the family of all possible such maps. For a fixed faulty set $F \in \mathcal{F}$ and $S \in \mathcal{S}$, the graph $G_{F,S}$ is a tuple $(\Pi_F, E_{F,S})$ where:

- $\Pi_F = \Pi - F$, i.e., the set of correct processes;
- Nodes p and q are connected with an edge *iff* their quorums $S(p)$ and $S(q)$ intersect in a correct process, i.e., $(p, q) \in E_{F,S} \Leftrightarrow S(p) \cap S(q) \not\subseteq F$.

▶ **Example 1.** Consider a system of four processes, where $\Pi = \{p_1, p_2, p_3, p_4\}$, $\mathcal{F} = \{\{p_3\}\}$, and the individual quorum systems are:

$$\mathcal{Q}(p1) = \{\{p_1, p_2, p_3\}\} \qquad \mathcal{Q}(p2) = \{\{p_1, p_2\}, \{p_2, p_4\}\}$$

$$\mathcal{Q}(p3) = \{\{p_1, p_2, p_4\}\} \qquad \mathcal{Q}(p4) = \{\{p_2, p_4\}, \{p_3, p_4\}\}$$

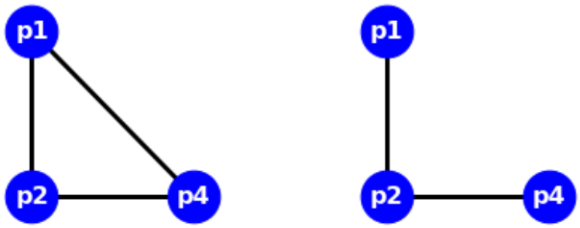


Figure 1 Graph structures of Example 1: G_{F,S_1} and G_{F,S_2} respectively.

Consider an execution with $F = \{p_3\}$, the set of correct processes Π_F is $\{p_1, p_2, p_4\}$. Let $S_1 \in \mathcal{S}$ be a quorum map for $\mathcal{Q}$ such that $S_1(p_1) = \{p_1, p_2, p_3\}$, $S_1(p_2) = \{p_1, p_2\}$ and $S_1(p_4) = \{p_2, p_4\}$. Every pair of nodes in the resulting graph G_{F,S_1} have quorums intersecting in $p_2 \in \Pi_F$, resulting in a fully connected graph. Now let $S_2 \in \mathcal{S}$ be a quorum map such that $S_2(p_1) = \{p_1, p_2, p_3\}$, $S_2(p_2) = \{p_2, p_4\}$ and $S_2(p_4) = \{p_3, p_4\}$. Since $S_2(p_1) \cap S_2(p_4) \subseteq F$, the resulting graph G_{F,S_2} has a missing edge. Figure 1 depicts G_{F,S_1} and G_{F,S_2}.

Inconsistency number. We recall two useful definitions from graph theory: *Independent Set* and *Independence Number*.

▶ **Definition 2** (Independent Set). *A set $C \subseteq V$ is an* independent set *of $G = (V, E)$ iff no pair of nodes in C is adjacent, i.e., $\forall p, q \in C : (p, q) \notin E$. C is* maximum *iff for every independent set C' of G: $|C'| \leq |C|$.*

▶ **Definition 3** (Independence Number). *The* independence number *of G is the size of its maximum independent set(s).*

Given the pair $(\mathcal{Q}, \mathcal{F})$, we note $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$ the family of graphs including all possible $G_{F,S}$, where $F \in \mathcal{F}$ and $S \in \mathcal{S}$.

▶ **Definition 4** (Inconsistency Number). *The* inconsistency number *of $(\mathcal{Q}, \mathcal{F})$ is the highest independence number among all $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$. Formally, Let $\mu : \mathcal{G}_{\mathcal{Q},\mathcal{F}} \to N$ map each $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$ to its independence number $\lambda(\mathcal{G}_{\mathcal{Q},\mathcal{F}})$, then $\lambda(\mathcal{G}_{\mathcal{Q},\mathcal{F}}) = max(\{\mu(G_{F,S}) | G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}\})$.*

▶ **Example 5.** Coming back to Example 1, the graph G_{F,S_1} is fully connected, thus it has independence number 1. On the other hand, the maximum independent set in G_{F,S_2} is $\{p1, p4\}$, as a result, G_{F,S_2} has independence number 2. Now consider a pair $(\mathcal{Q}, \mathcal{F})$ where $\mathcal{Q}$ and $\mathcal{F}$ are the same as in Example 1 (with this assumption only p_3 may fail in any execution). Since $\forall S \in \mathcal{S}, \forall F \in \mathcal{F} : S(p_1) \cap S(p_2) \not\subseteq F$, it is easy to see that no graph has independence number higher than 2 in $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$, thus the inconsistency number of $(\mathcal{Q}, \mathcal{F})$ is 2.

Computing inconsistency parameters. A straightforward approach to find the inconsistency number of $(\mathcal{Q}, \mathcal{F})$ consists in computing the independence number of all graphs $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$. The problem of finding the *maximum independent set* of a graph, and consequently its independence number, is the *maximum independent set problem* [39], known to be *NP-complete* [35]. Also, the number of graphs in $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$ may exponentially grow with the number of processes. However, as the graphs might have similar structures (for example, the same quorums for some processes may appear in multiple graphs), in many practical scenarios, we should be able to avoid redundant calculations and reduce the overall computational costs, as we show for the uniform model.

Table 1 Inconsistency numbers for classical BQS with 100 processes.

Faulty processes	0–33	34–50	51–55	56–58	59–60	61	62	63	64	65	66
Inconsistency Number	1	2	3	4	5	6	7	9	12	17	34

Inconsistency in the uniform model. Centralized quorum systems generate graphs that are similar in structure and are therefore easier to analyse. Given a uniform quorum system $\mathcal{Q}_u$, we show how to calculate the inconsistency number of $(\mathcal{Q}_u, \mathcal{F}_u)$, where $\mathcal{Q}_u$ and $\mathcal{F}_u$ include every subset of processes with sizes q and $\leq f$ respectively, in which $f < q$.

▶ **Theorem 6.** *Let $(\mathcal{Q}_u, \mathcal{F}_u)$ be a uniform quorum system with n processes, quorums of size q and where at most f processes might fail. The inconsistency number of $(\mathcal{Q}_u, \mathcal{F}_u)$ is $\lfloor \frac{n-f}{q-f} \rfloor$.*

Proof. Fix any $F \in \mathcal{F}_u$ of size f. Let $G_{F,S} \in \mathcal{G}_{\mathcal{Q}_u,\mathcal{F}_u}$ be a graph whose independence number is the highest, and let $C_{max} = \{p_1, ..., p_m\}$ be a maximum independent set in $G_{F,S}$. Let $cor(Q)$ denote the number of correct processes in a quorum Q and let $Q_i = S(p_i)$. It follows that $cor(Q_1) + ... + cor(Q_m) \leq n - f$, since the quorums $Q_1, ..., Q_m$ have no correct processes in common. We can then build a graph $G_{F,S'}$ with an independent set $C' = \{p_1, ..., p_m\}$, where $\forall p_i \in C' : F \subseteq S'(p_i)$, that is, the quorum for every $p_i \in C'$ includes all faulty processes. Indeed, it suffices to choose $S'(p_i)$ as any $q - f$ correct processes from $S(p_i)$, in addition to the f faulty processes. As there can be at most $k_{max} = \lfloor \frac{n-f}{q-f} \rfloor$ disjoint sets of $q - f$ correct processes, we conclude that the maximum value m can reach is k_{max}. ◀

▶ **Example 7.** A classical *Byzantine quorum system* (BQS) uses quorums of size $q = 2n/3 + 1$. It is typically assumed that $f < n/3$ processes. As Theorem 6 implies, the inconsistency number of this range is 1, and it grows with f. Table 1 illustrates how the inconsistency number varies with the number of faulty processes in a system with 100 processes.

4 Asset Transfer System

In this section, we define the problem of k-spending asset transfer (k-AT) and describe a protocol that solves k-AT in a given trust model $(\mathcal{Q}, \mathcal{F})$, where k is the inconsistency number of $(\mathcal{Q}, \mathcal{F})$.

4.1 Preliminaries

Transactions. A transaction $tx \in \mathcal{T}$ is a tuple $(s, \tau, I, data)$, where s is the process identifier of the *issuer*, $\tau : \Pi \to \mathbb{Z}_0^+$ is the *output map* and $I \subseteq \mathcal{T}$ is the set of *input* transactions, tx is called *outgoing from* s and *incoming to* every p such that $tx.\tau(p) > 0$. Also, every transaction in $tx.I$ must be *incoming to* $tx.s$. Finally, *data* is a bit-string attached to the transaction which contains some arbitrary information.

We use the function $inValue\colon \mathcal{T} \to \mathbb{Z}_0^+$ to denote the sum of the amount sent to s by the transaction inputs, i.e., $inValue(tx) = \sum_{tx' \in tx.I} tx'.\tau(s)$. The function $outValue\colon \mathcal{T} \to \mathbb{Z}_0^+$ denotes the total amount spent in a transaction, i.e., $outValue(tx) = \sum_{p \in \Pi} tx.\tau(p)$. A transaction tx is *valid iff* $outValue(tx) > 0$ and $outValue(tx) = inValue(tx)$. Since the issuer of tx might not send the entire value of its inputs to other processes, we allow the remaining amount to be transferred back to the issuer in its output map. We assume from this point on that each transaction in a history is signed by its issuer.

We assume that the total stake is initially distributed in a special transaction $tx_{init} = (\bot, \tau_{init}, \emptyset)$. The total stake of the system is therefore $\sum_{p \in \Pi} \tau_{init}(p)$.

Two distinct transactions tx and tx' *conflict* if they are issued by the same process and share some input, i.e., $(tx.s = tx'.s) \wedge (tx.I \cap tx'.I \neq \emptyset)$.

Transaction histories. A set of transactions $T \subseteq \mathcal{T}$ is called a *transaction history*. T generates a directed graph, where each $tx \in T$ is a node and directed edges are drawn to tx from its inputs. Let $tx, tx' \in T$, if tx is reachable from tx' in this graph (i.e., there is a path from tx' to tx), we say tx depends on tx'. A transaction history T is *well-formed iff*:

- (T-Validity) $tx_{init} \in T \wedge \forall tx \in T, tx \neq tx_{init} : tx$ is valid;
- (Completeness) $\forall tx \in T, \forall tx' \in tx.I : tx' \in T$;
- (No-Conflict) $\forall tx, tx' \in T : tx$ and tx' do not conflict;
- (Cycle-Freedom) $\forall tx, tx' \in T : tx$ depends on $tx' \Rightarrow tx \notin tx'.I$.

We only consider well-formed histories from this point on. The function $balance_T : \Pi \to \mathbb{Z}$ applied to a transaction history T determines the *balance* of each process w according to T: $balance_T(w)$ is the difference between the sum of transfers to w and the sum of transfers issued by w, i.e.,

$$balance_T(w) = \sum_{tx \in T} tx.\tau(w) \; - \sum_{tx \in T, tx.s = w} outValue(tx)$$

▶ **Proposition 8.** *Given a well-formed history T, for every process w, $balance_T(w) \geq 0$.*

Proof. Let $\sum_{tx \in T} tx.\tau(w)$ be the *incoming stake* to w and let $\sum_{tx \in T} outValue(tx)$, with $tx.s = w$, be the *outgoing stake* from w. Assume that $\text{balance}_T(w) < 0$, then the outgoing stake is greater than the incoming stake. The initial transaction tx_{init} may only send funds to w, and since every other transaction $tx \in T$ is valid, tx must include inputs with enough funds to cover $outValue(tx)$. From *Completeness*, for every transaction tx appearing in the sum of the outgoing stake, its inputs $tx' \in tx.I$ also appear in the sum of the incoming stake. Therefore, the only remaining way w can spend more stake than it received is to use an input more than once, which is prevented by *No-Conflict*. ◀

Although a well-formed history has no conflicting transactions, there may exist conflicts among distinct well-formed histories. Consider a collection of well-formed transaction histories Γ, a process r, and tx an incoming transaction to r. Let $I^r_{tx} \subseteq \mathcal{T}$ be the set of outgoing transactions from r, each $tx' \in I^r_{tx}$ including tx in its input and appearing in some $T_i \in \Gamma$,

$$I^r_{tx} = \{tx' \mid \exists T_i \in \Gamma : (tx' \in T_i) \wedge (tx \in tx'.I) \wedge (tx'.s = r)\}.$$

Let $|I^r_{tx}| = k$, we say that process r *k-spends* tx in Γ. In other words, a process k-spends if it issued k distinct transactions appearing in Γ using the same input.

▶ **Definition 9** (Spending Number). *Let Γ be a collection of well-formed histories. The* spending number *of Γ, noted $\gamma(\Gamma)$, is the highest amount of times an input is spent by the same process in Γ. Formally,*

$$\gamma(\Gamma) = max(\{|I^r_{tx}| \mid \forall r \in \Pi, \forall tx \text{ incoming to } r\}).$$

Note that, by definition, the spending number of Γ cannot exceed $|\Gamma|$.

4.2 Problem Statement

Every process $p \in \Pi$ maintains a *local history* T_p, where p *accepts* tx when it adds tx to T_p.

Ideally, we want local histories of correct processes to eventually converge. But this may not always be possible, as our specification allows for multiple spending: correct process may accept conflicting transactions. Therefore, we also introduce an accountability mechanism, expressed in the form of accusation histories.

Formally, an *accusation* is a tuple (AC, P) consisting of a set of processes $AC \subseteq \Pi$ and a *proof of misbehavior* P for every process in AC. (AC, P) can be independently verified by a third party through the function *verify-acc:* $(2^{\Pi} \times \mathcal{P}) \to \{true, false\}$. Technically, for each process $p \in AC$, the proof P must contain a set of conflicting transactions $tx_1, ..., tx_\ell$ signed by p. We say that the accusation (AC, P) *refers* to $tx_1, ..., tx_\ell$.

Every process p is also expected to maintain a local accusation history A_p, where each element in A_p is an accusation tuple. The *k-spending asset transfer abstraction* receives inputs of the form *transfer(tx)* and produces updates to the local histories T_p and A_p.

Consider a run of a *k-spending asset transfer protocol* (k-AT) in a trust model $(\mathcal{Q}, \mathcal{F})$ with a fixed faulty set $F \in \mathcal{F}$. Let $T_p(t)$ and $A_p(t)$ denote the transaction history and accusation history of process p at time t, respectively. Let $\Gamma(t)$ denote the collection of local histories of correct processes at time t. Then the run must satisfy:

Validity If a correct process issues a transaction tx, then every live correct process p eventually adds tx to T_p, or adds an accusation to A_p referring to some transaction on which tx depends.

k-Spending For all $t \geq 0$, the spending number of $\Gamma(t)$ is bounded by k, i.e., $\gamma(\Gamma(t)) \leq k$.

Eventual Conviction If correct processes p and q add conflicting transactions tx to T_p and tx' to T_q respectively, then they eventually add an accusation referring to tx and an accusation referring to tx' to A_p and A_q.

Accuracy For all $t \geq 0$ and (AC, P) in $A_p(t)$: *verify-acc*$(AC, P) = true$. Moreover, *verify-acc*(AC, P) returns *true* if and only if $AC \subseteq F$.

Agreement If a correct process p adds an accusation (AC, P) to A_p, then every correct process eventually adds (AC, P) to its accusation history.

Integrity If $tx.s$ is correct, a correct process p adds tx to T_p only if $tx.s$ previously issued tx.

Monotonicity The accusation history of correct processes grows monotonically, i.e., for all p correct and $t \leq t'$, $A_p(t) \subseteq A_p(t')$;

Termination If a correct process p adds a transaction tx to T_p, then every live correct process q eventually adds tx to T_q or an accusation referring to tx (or some transaction on which tx depends) to A_q.

4.3 k-Spending Asset Transfer Protocol

The pseudo-code of our *k-spending asset transfer protocol* is presented in Algorithms 1 and 2. In the protocol, a process accepts a transaction only after hearing from a (local) quorum, and after all of the transaction's inputs have already been accepted.

Local Variables. Variables *echoes*, *usedInp* and *pending* are used in a broadcast stage of the algorithm. The array *echoes* stores received transactions echoed by other processes. In *usedInp*, p_i stores all transactions it has witnessed to be used as inputs, while in *pending* it stores transactions with signatures from at least a quorum that have not yet been added to the history. The remaining variables are: p_i's transaction history *trHist*, p_i's accusation history *acHist*, and *signedReq*, an array with sets of tuples (tx, σ), where σ is a signature for tx from $tx.s$.

The complete algorithm consists of three main blocks: the *broadcast* block, the *acceptance* block and the *accountability* block. In the following, we give a detailed description on how each block operates.

Broadcasting transactions. In order to issue a transaction, p_i specifies a transaction tx and invokes the operation $transfer(tx)$ (we assume that transactions issued by correct processes are always valid). Process p_i then creates a signature σ for tx and sends them in a REQ message to every process in the system. Upon receiving REQ with tx, p_i stores the signed transaction in $signedReq$. If none of tx's inputs are in $usedInp[tx.s]$, p_i echoes the original signed request with the issuer's signature and adds the inputs of the transaction to $usedInp[tx.s]$. A message whose signature does not match its sender is ignored.

Each time a new $ECHO$ is received from p_j for a transaction tx, p_i stores the echoed transaction in $echoes[p_j]$ and follows the same steps as when receiving a REQ message. When "enough" echoes are collected for the same transaction tx, and if tx is neither in $pending$ nor $trHist$, it is added to $pending$.

Accepting transactions. After going through the broadcast phase and adding tx to $pending$, the function $ready(tx)$ is used to verify whether the addition of tx to $trHist$ results in a well-formed history. If *T-Validity*, *Completeness*, and *No-Conflict* still hold, p_i adds tx to $trHist$ and removes it from $pending$ (as later shown in Lemma 10, Cycle-Freedom is guaranteed by construction since a single transaction is added at a time).

Treating accusations. Since p_i keeps track of every received request (tx, σ) in $signedReq$ (either coming directly from a REQ message or from an $ECHO$), it can construct a proof of misbehavior after receiving signed conflicting transactions. The proof here consists of a pair (tx, σ_j) and (tx', σ'_j) containing distinct transactions from p_j whose inputs have a non-empty intersection. An accusation (AC, P) is created using p_j's identifier and the proof. If it is a new accusation, p_i adds (AC, P) to $acHist$ and sends it to every process in an ACC message. The same steps are followed once an ACC is received with a verifiable accusation tuple (AC, P).

Correctness. Consider executions of Algorithms 1 and 2 assuming trust model $(\mathcal{Q}, \mathcal{F})$ with inconsistency number k_{max}. Let $F \in \mathcal{F}$ be a corresponding faulty set.

▶ **Lemma 10.** *The history T_p of a correct process p is well-formed, i.e., satisfies the properties of T-Validity, Completeness, No-Conflict and Cycle-Freedom.*

Proof. The default value of $trHist$ is $\{tx_{init}\}$, which is well-formed by definition. Now assume that at some point $trHist$ is well-formed. Before adding a new transaction tx at line 21, p invokes $ready(tx)$ to check whether tx is valid and that the resulting history satisfies *No-Conflict* and *Completeness* (lines 34 to 38). By construction, $trHist$ is also *Cycle-Free*: suppose $\{tx\} \cup trHist$ creates a cycle, that is, $\exists tx' \in \{tx\} \cup trHist$ on which tx depends and $tx \in tx'.I$. This is clearly not possible: since $trHist$ satisfies *Completeness*: $\forall tx'' \in tx'.I : tx'' \in trHist$, but $tx \notin trHist$, a contradiction. ◀

▶ **Lemma 11** (k-Spending). *At any time t, the spending number of $\Gamma(t)$ is bounded by k_{max}.*

Proof. Let $r \in F$ and tx be an incoming transaction to r. Suppose r spends tx k times in $\Gamma(t)$, with $k > k_{max}$. We assume, without loss of generality, that r is the process that multiple spent the maximal number of times in $\Gamma(t)$, that is, $\gamma(\Gamma(t)) = k$. We make the following observations about the algorithm:

Algorithm 1 k-Spending Asset Transfer System: code for process p_i part 1.

Local Variables:
$echoes \leftarrow [\emptyset]^N$; /* Array containing sets of received echoes */
$usedInp \leftarrow [\emptyset]^N$; /* Array of inputs used by each process */
$pending \leftarrow \emptyset$; /* Set of transactions waiting to be accepted */
$trHist \leftarrow \{tx_{init}\}$; /* Transaction History of p_i */
$signedReq \leftarrow [\emptyset]^N$; /* An array of set of pairs transaction-signature */
$acHist \leftarrow \emptyset$; /* Accusation history of p_i */

behavior:
Ignore messages with invalid signatures;

1 **operation** *transfer*(tx)**:**
2 $\sigma \leftarrow sign(self, tx)$;
3 send $\langle REQ, tx, \sigma \rangle$ to all $p \in \Pi$;

4 **upon** *receiving* $\langle REQ, tx, \sigma_j \rangle$ *from* p_j**:**
5 $signedReq[tx.s] \leftarrow signedReq[tx.s] \cup \{(tx, \sigma_j)\}$;
6 **if** $tx.I \cap usedInp[tx.s] = \emptyset$ **then:**
7 $usedInp[tx.s] \leftarrow usedInp[tx.s] \cup tx.I$; /* Stores used inputs */
8 $\sigma \leftarrow sign(self, tx)$;
9 send message $\langle ECHO, (tx, \sigma_j), \sigma \rangle$ to all $p \in \Pi$;

10 **upon** *receiving* $\langle ECHO, (tx, \sigma_s), \sigma_j \rangle$ *from* p_j**:**
11 $echoes[p_j] \leftarrow echoes[p_j] \cup \{tx\}$;
12 $signedReq[tx.s] \leftarrow signedReq[tx.s] \cup \{(tx, \sigma_s)\}$;
13 **if** $tx.I \cap usedInp[tx.s] = \emptyset$ **then:**
14 $usedInp[tx.s] \leftarrow usedInp[tx.s] \cup tx.I$;
15 $\sigma \leftarrow sign(self, tx)$;
16 send message $\langle ECHO, (tx, \sigma_s), \sigma \rangle$ to all $p \in \Pi$;

17 **upon** *receiving echoes for tx from a quorum* $Q_i \in \mathcal{Q}(p_i)$**:**
18 **if** $tx \notin trHist \wedge tx \notin pending$ **then:**
19 $pending \leftarrow pending \cup \{tx\}$; /* Collected enough signatures for tx */

Algorithm 2 k-Spending Asset Transfer System: code for process p_i part 2.

20 **upon** *existing* $tx \in pending$ *such that* $ready(tx) =$ ***true*** **:**
21 $trHist \leftarrow trHist \cup \{tx\}$; /* Adds transaction to history */
22 $pending \leftarrow pending / \{tx\}$;

23 **upon** *existing distinct* tx *and* tx' *in* $signedReq[p_j]$ *such that* $tx.I \cap tx'.I \neq \emptyset$**:**
24 $ev_1 \leftarrow (tx, \sigma_j)$; /* Evidences of misbehavior */
25 $ev_2 \leftarrow (tx', \sigma'_j)$;
26 $accusation \leftarrow (\{p_j\}, \{ev_1, ev_2\})$; /* $AC = \{p_j\}$, $P = \{ev_1, ev_2\}$ */
27 **if** $accusation \notin acHist$ **then:**
28 $acHist \leftarrow acHist \cup \{accusation\}$; /* Adds accusation to history */
29 send $\langle ACC, accusation \rangle$ to all $p \in \Pi$;

30 **upon** *receiving* $\langle ACC, accusation \rangle$ *from* p_j**:**
31 **if** $accusation \notin acHist \wedge$ *verify-acc*$(accusation)$ **then:**
32 $acHist \leftarrow acHist \cup \{accusation\}$;
33 send $\langle ACC, accusation \rangle$ to all $p \in \Pi$;

34 **operation** *ready*(tx)**:**
35 $c_1 \leftarrow \forall tx' \in tx.I : tx' \in trHist$; /* *Completeness* */
36 $c_2 \leftarrow$ **true** *iff* tx is valid; /* *T-Validity* */
37 $c_3 \leftarrow \forall tx' \in trHist : (tx'.s = tx.s) \Rightarrow (tx'.I \cap tx.I = \emptyset)$; /* *No-Conflict* */
38 **return** $c_1 \wedge c_2 \wedge c_3$;

1. A correct process p adds a transaction tx' to its history only if it received *ECHO* messages for tx' from every process in a quorum $Q \in \mathcal{Q}(p)$ (guard in line 17).
2. p checks if any input of a received transaction is already in *usedInp* before echoing it (lines 6 and 13), and if it sends *ECHO* for a transaction, p adds all of its inputs to *usedInp* (lines 7 and 14). Therefore, p can only send *ECHO* for a single transaction from r that has tx as an input.

Let correct processes p_i and p_j accept conflicting transactions tx_i and tx_j from r after receiving echoes from $Q_i \in \mathcal{Q}(p_i)$ and $Q_j \in \mathcal{Q}(p_j)$, respectively. From (2) above, we conclude that $Q_i \cap Q_j \subseteq F$, otherwise a correct process in the intersection would have echoed two different transactions sharing some input(s) from r.

Since r k-spends tx in $\Gamma(t)$, there exists $p_1, ..., p_k$ correct that accepted, respectively, conflicting $tx'_1, ..., tx'_k$ from r using tx as input. Now let $Q_1 \in \mathcal{Q}(p_1), ..., Q_k \in \mathcal{Q}(p_k)$ be the quorums each process received echoes from before accepting the transactions. We can build a quorum map S satisfying $S(p_i) = Q_i$ for $i = 1, ..., k$, and a graph $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$ of which $C = \{p_1, ..., p_k\}$ is an independent set, since from (1) and (2) above: $\forall p_i, p_j \in C, i \neq j : S(p_i) \cap S(p_j) \subseteq F$. However, k_{max} is the inconsistency number of $(\mathcal{Q}, \mathcal{F})$, which means there cannot be a graph $G_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$ with an independent set of size $k > k_{max}$, a contradiction. ◀

▶ **Lemma 12** (Eventual Conviction). *If correct processes p and q add conflicting transactions tx to T_p and tx' to T_q, respectively, then they eventually add an accusation referring to tx and an accusation referring to tx' to A_p and A_q respectively.*

Proof. Before accepting tx and tx', p and q received echoes for tx (in p's case) and tx' (in q's case), storing the original signed requests in their local *signedReq* (line 12). There are two possible scenarios for each process (for simplicity, we only describe them for p): p echoed tx before adding it to T_p, or p did not echo tx. If p echoed tx, then q will eventually receive the echo with a signed request for tx from p, which allows q to construct and relay an accusation for $tx.s$ (in lines 23 to 29) using this request together with the one for tx' already stored in q's *signedReq* (e.g. assigning the request for tx to ev_1 at line 24 and the request for tx' to ev_2 at line 25). Eventually p will receive an *ACC* message from q containing this accusation and will add it to its accusation history.

Now if p did not echo tx, then it must have echoed for another conflicting transaction tx'', which means p can construct an accusation using the respective signed requests for tx and tx'' as described above. This accusation is sent to every process and is eventually received by q, which then adds it to *acHist*. Ultimately, both p and q add accusations referring to tx and tx' to their accusation histories. ◀

▶ **Lemma 13** (Termination). *If a correct process p adds a transaction tx to T_p, then every live correct process q eventually adds tx to T_q or an accusation referring to tx (or some transaction on which tx depends) to A_q.*

Proof. Recall that a process is live if it has a quorum composed of only correct processes.

We first show the following: If a correct process adds a transaction tx to *pending*, then every live correct process eventually does so or adds an accusation referring to tx to *acHist*.

Let p be a correct process that adds tx to its *pending* after receiving echoes for tx from a quorum. There are two cases to consider, depending on whether p previously echoed tx or not.

If p did not echo tx, then it echoed a conflicting tx' and built an accusation (AC, P) with the original requests for tx and tx' (lines 23 to 26). Then, p adds the accusation to its *acHist* and sends (AC, P) to all processes. Every correct process eventually receives the accusation and also adds it to *acHist*.

Suppose now that p echoed tx. If no process sent *ECHO* or *REQ* for a conflicting transaction, then every correct process eventually receives and echoes tx. If a correct process q is live, it will eventually receive enough echoes and add tx to *pending*. On the other hand, if a process in q's live quorum had echoed a conflicting transaction, q will receive the conflicting requests, build an accusation (AC, P) referring to tx and tx' and send it to all processes. Then, as described previously, every correct process will eventually add (AC, P) to *acHist*.

Now suppose p also adds tx to its *trHist*. We make the following observations about the algorithm: before being added *trHist*, any transaction tx' is first added to *pending* (guard in line 20). Also, from Lemma 10 every transaction on which tx' depends must have been previously added to *trHist*. Let *deps(tx)* include tx and every transaction on which tx depends. It follows that p previously added every $tx' \in$ *deps(tx)* to *pending*. The following three cases are then possible for a live correct process q:

1. q eventually adds every $tx' \in$ *deps(tx)* to its *pending*. If no transaction in *trHist* conflicts with them, q adds every such tx' to *trHist*.
2. q has already added a transaction to *trHist* that conflicts with some $tx' \in$ *deps(tx)*. In this case, it received conflicting requests. q will then build and send everybody an accusation including the signed requests for the respective transactions.
3. q never adds one (or more) $tx' \in$ *deps(tx)* to *pending*, in which case, as previously shown, q eventually adds an accusation referring to tx' to *acHist*.

Therefore, if a correct process p adds a transaction tx to *trHist* and a live correct process q is never able to do so, then q eventually adds an accusation to *acHist* referring to tx or some transaction on which tx depends. ◀

▶ **Lemma 14** (Validity). *If a correct process issues a transaction tx, then every live correct process p eventually adds tx to T_p, or adds an accusation to A_p referring to some transaction on which tx depends.*

Proof. If correct process p sends a request for tx, eventually every correct process echoes tx and every live correct process adds tx to its *pending*. Since p is correct, it will not send conflicting requests, thus no accusation referring to tx can be produced. Also, p must have previously added every transaction on which tx depends to *trHist*, which from Lemma 13, if a live correct process q does not add said transactions to *trHist* (and consequently tx), q eventually adds an accusation to *acHist* referring to some transaction on which tx depends. ◀

▶ **Theorem 15.** *Consider the trust model $(\mathcal{Q}, \mathcal{F})$ with inconsistency number k_{max}. Algorithms 1 and 2 implement k_{max}-spending asset transfer abstraction.*

Proof. Lemma 10 shows that correct processes always maintains well-formed local transaction histories. The k_{max}-*Spending*, *Eventual Conviction*, *Termination* and *Validity* properties are shown in Lemmas 11 to 14.

Accuracy, *Monotonicity*, *Agreement* and *Integrity* are immediate. A correct process adds an accusation (AC, P) to *acHist* only if it can verify that messages for conflicting transactions in P were indeed signed by the processes in AC (*Accuracy*). The set *acHist* may only grow with time (*Monotonicity*). Moreover, once a correct process adds an accusation to its *acHist*, it sends the accusation to every other process. This accusation is eventually received by every correct process, which verifies and adds it to *acHist* (*Agreement*). Finally, the signature of a correct process for a transaction request cannot be forged (*Integrity*). ◀

5 Relaxed Broadcast Abstraction and Lower Bounds

In this section, we show that the inconsistency number of $(\mathcal{Q}, \mathcal{F})$ is optimal for k-AT, by relating the problem to the fundamental *broadcast abstraction*. The abstraction exports one operation *broadcast(m)* and enables a callback *deliver(m)*, for m in a value set $\mathcal{M}$. We assume that each broadcast instance has a dedicated *source*, i.e., the process invoking the broadcast operation.

We now describe *k-Consistent Broadcast*, first introduced in [5]. Given a trust model $(\mathcal{Q}, \mathcal{F})$, in every execution with a fixed $F \in \mathcal{F}$, a k-Consistent Broadcast protocol ensures the following properties:

- (Validity) If the source is correct and broadcasts m, then every *live* correct process eventually delivers m.
- (k-Consistency) Let M be the set of values delivered by correct processes, then $|M| \leq k$.
- (Integrity) A correct process delivers at most one value and, if the source p is correct, only if p previously broadcast it.

This protocol is a generalized version of an abstraction known as *Consistent Broadcast* [9]. *Validity* in Consistent Broadcast guarantees that a broadcast value is delivered by every correct process. Also, correct processes cannot deliver different values. Note that if every correct process is live and $k = 1$, then k-CB implements Consistent Broadcast.

5.1 Lower bound for k-Consistent Broadcast

We restrict our attention to *quorum-based protocols*, initially introduced in the context of consensus algorithms [32]. Intuitively, in a quorum-based protocol, a process p should be able to make progress if the members in one of its quorums $Q \in \mathcal{Q}$ appear correct to p. This should hold even if the actual set of correct processes in this execution is different from Q. Formally, we make the following assumption about algorithms implementing k-CB:

- (Local Progress) For all $p \in \Pi$ and $Q \in \mathcal{Q}(p)$, there is an execution in which only the source and processes in Q take steps, p is correct, and p delivers a value.

Consider a trust model $(\mathcal{Q}, \mathcal{F})$ and let k_{max} be its inconsistency number, then:

▶ **Theorem 16.** *No algorithm can implement k-CB such that $k < k_{max}$.*

Proof. Let $G_{F,S}$ be the graph generated over fixed $F \in \mathcal{F}$ and $S \in \mathcal{S}$ and $C = \{p_1, ..., p_{k'}\}$ an independent set in $G_{F,S}$ of size k'. We proceed to show that there exists an execution where k' different values are delivered by processes in C. Let r be the source, by the definition of *Local Progress*, there exists an execution Σ_i for each $p_i \in C$ where $part(\Sigma_i) = \{r\} \cup S(p_i)$, in which p_i delivers a value m_i. Now assume that r is faulty, we can build an execution Σ such that all the correct processes in $part(\Sigma_i)$ take the same steps in Σ as in Σ_i, and all the faulty processes behave to them (send the same messages) the same as in execution Σ_i. For a correct process in $part(\Sigma_i)$, Σ is then indistinguishable from Σ_i, and thus $p_i, ..., p_{k'}$ must deliver $m_1, ..., m_{k'}$ respectively.

Now let $G'_{F,S} \in \mathcal{G}_{\mathcal{Q},\mathcal{F}}$ be a graph whose independence number is k_{max}, there exists an independent set C_{max} of size k_{max} in $G'_{F,S}$. As shown above, it is possible to build an execution where k_{max} processes ($k_{max} \geq k'$) deliver k_{max} distinct values before any correct process is able to identify the misbehavior. ◀

Intuitively, if two correct processes have quorums that do not have a correct process in the intersection, they might deliver distinct values before noticing any misbehavior in the execution. Within an independent set, the quorums of every pair of nodes do not intersect in a correct process, and k_{max} represents the highest possible independent set in $\mathcal{G}_{\mathcal{Q},\mathcal{F}}$, thus establishing the lower bound for k-CB.

5.2 Relating k-Spending Asset Transfer and k-Consistent Broadcast

We show now that having a protocol implementing k-AT, one implement k-CB, which implies that the lower bound established in Theorem 16 also holds for k-AT.

▶ **Theorem 17.** *k-AT can be used to implement k-CB.*

Proof. Suppose that we have a protocol implementing k-AT. First, we let tx_{init} assign some funds to the source p, and assume that p broadcasts a message to other processes by means of encoding it in a transaction's *data*.

Therefore, to broadcast a message m, p issues a transaction $tx = (p, \tau, \{tx_{init}\}, m)$. Whenever a correct process q adds tx to T_q, it issues *deliver(m)*.

If p is correct, every live correct process eventually delivers it, that is k-AT *Validity* implies k-CB *Validity*. Moreover, since processes do not have knowledge of $\mathcal{F}$, an algorithm implementing k-AT must guarantee *Validity* for an arbitrary fault scenario $F \in \mathcal{F}$. As such, for particular source p, process p_i and quorum $Q_i \in \mathcal{Q}(p_i)$, if F is such that every process other than $p \cup Q_i$ is faulty, then there must be an execution in which only these processes take step and p_i accepts the transaction from p, implying the *Local Progress* assumed in k-CB protocols.

From the *k-Spending* property, up to k conflicting transactions issued by p with tx_{init} as input can be accepted by correct processes. Thus, at most k distinct messages might be "delivered", which implies *k-Consistency*. Trivially, k-AT *Integrity* implies k-CB *Integrity*. ◀

Theorems 16 and 17 imply that Algorithms 1 and 2 implement k-AT with optimal k.

6 Related Work

Damgård et al. [16] appear to be the first to consider the decentralized trust setting. They introduced the notion of *aggregate adversary structure* $\mathcal{A}$: each node is assigned with a collection of subsets of nodes that the adversary might corrupt at once. In this model, assuming *synchronous* communication, they discuss solutions for broadcast, verifiable secret sharing and multiparty computation.

Ripple [37] and Stellar [34], conceived as *open* payment systems, use decentralized trust as an alternative to *proof-of-work*-based protocols [36, 41]. In the Ripple protocol, each participant expresses its trust assumptions in the form of a *unique node list* (UNL), a subset of system members. To accept a transaction, a node needs acknowledgement from a set of at least 80% of its UNL (which can be seen as a quorum). The Ripple white paper [37] assumes that up to 20% of members in a UNL are Byzantine, stating that an overlap of at least 20% between every pair of UNLs in enough to prevent *forks*. Later analyses suggest this overlap to be more than 40% [3] without Byzantine faults, and more than 90% with the same original assumptions [13] (up to 20% Byzantine members in a UNL). Chase and MacBrough [13] also provide an example in which liveness of the protocol is violated even with 99% of overlap.

Stellar consensus protocol [34] uses a *Federated Byzantine Quorum System* (FBQS). A quorum Q in the FQBS is a set that includes a *quorum slice* (a trusted subset of members) for every member in Q. Correctness of Stellar depends on the individual trust assumptions,

and stronger properties are guaranteed for nodes trusting the "right guys", which are in so called *intact sets*. García-Pérez and Gotsman [19] treated Stellar consensus formally, by relating it to Bracha's Broadcast Protocol [6], build on top of a FBQS. Their analysis has been later extended [20] to a variant of state-machine replication protocol that allows *forks*, where disjoint intact sets may maintain different copies of the system state.

Losa et al. [32], Sheff et al. [38], and Cachin [8] investigate more general formalizations of decentralized trust. Cachin and Tackmann [11] model trust assumptions via an *asymmetric fail-prone system*, an array $[\mathcal{F}_1, ..., \mathcal{F}_n]$ of adversary structures (or fault models), where each $\mathcal{F}_i$ is chosen by p_i as its local fault model. For this model, they devise an *asymmetric Byzantine quorum system* (ABQS), an array of quorum systems $[\mathcal{Q}_1, ..., \mathcal{Q}_n]$ satisfying specific intersection and availability properties, so that certain problems, such as broadcast and storage, can be solved. Recently, Losa et al. [10] extended this formalism to *permissionless* settings where the processes may make assumptions about each others' trust models.

Losa et al. [32] introduced the notion of a *Personal Byzantine Quorum System* (PBQS), where every process chooses its quorums with the restriction that if Q is a quorum for a process p, then Q includes a quorum for every process $q' \in Q$. The PBQS model is then discussed in relations to Stellar consensus [34]. More precisely, they characterize the conditions on PBQS under which a *quorum-based* protocol (captured by our Local Progress condition) ensures that a well-defined subset of processes (a *consensus cluster*) can maintain safety and liveness of consensus.

In contrast, we allow the processes to directly choose their quorums, and we address the question of what is the "best" consistency a cryptocurrency can achieve within this trust model. The measure of consistency is quantified here as the spending number. In a way, unlike this prior work on decentralized trust, instead of searching for the weakest trust model that enables solutions to a given problem, we determine the "strongest" problem (in a specific class) that is possible to solve in a given model.

In the context of distributed systems, accountability has been proposed as a mechanism to detect "observable" deviations of system nodes from the algorithms they are assigned with [25, 24, 26]. Recent proposals [14, 17] focus on *application-specific* accountability that only heads for detecting misbehavior that affects correctness of the problem to be solved, e.g., consensus [14] or lattice agreement [17]. Our k-AT algorithm generally follows this approach, except that it implements a relaxed form of asset transfer system, but detects violations that affect correctness of the stronger, conventional asset transfer abstraction [22].

7 Discussion and Future Work

Generalizing the inconsistency measure. Our notion of the inconsistency number of a trust model $(\mathcal{Q}, \mathcal{F})$ serves to quantify the amount of times a process can spend the same input in our cryptocurrency implementation (or the number of distinct values that can be delivered by correct processes in our broadcast abstraction). A natural variation of this question is to explore the conditions on a trust model that are necessary and sufficient to implement a cryptocurrency which bounds the number of copies the same asset can maintain in a system. Note that our notion of k-spending is more relaxed, as it only bounds the number of times the same input transaction can be used by a process.

It is very interesting to apply "inconsistency metrics" for solving other, more general problems in the decentralized trust setting. Consider for example *State Machine Replication* (SMR) protocols [31, 12]. In these protocols, correct processes agree on a global history of concurrently applied operations, and thus witness the same evolution of the system state.

One way to relax consistency guarantees of SMR protocols in decentralized trust settings is to bound the number k of diverging histories (the maximum degree of the fork). The question is then how to relate k to the trust model $(\mathcal{Q}, \mathcal{F})$.

Reconfiguration of inconsistent states. Our k-AT abstraction provides the application with the history of valid transactions and a record of misbehaving parties. An important question is left open: once correct processes accept conflicting transactions and accusations against Byzantine processes are raised, what is next? Is there a way to render the system back to a consistent state? Although there is no general answer to these questions – it comes down to what better suits the application – we point out some of the suitable strategies.

A natural response to this is to *reconfigure* both the trust model and the states of the processes, in order to achieve some desired level of consistency. The immediate use of an accusation (AC, P) is to rearrange the trust assumptions by evicting the misbehaving parties AC from the system. For example, the application might use the accusation history to suggest new (improved) quorum systems to system members. One may hope to deploy recently developed asynchronous reconfiguration schemes [1, 29, 30].

When it comes to reconfiguration of the system states, we face a more challenging task. Indeed, some correct processes may have already used "compromised" (multiply spent) assets in their transactions. "Merging" conflicting histories into a consistent global state might affect the stake distribution, which can be hard to resolve without changing the application semantics. We present two strategies that make use of the transactions identified in accusation proofs. The first approach, alluding to real financial systems, is to let them keep (and use as input) accepted conflicting transactions after the misbehaving parties are excluded from the system. This can have implications on the total stake of the system: depending on how much stake was spent, the total system stake may grow. Once multi-spending party may get negative balance in its accounts, and could be "frozen", i.e., forbidden to issue new transactions until the balance turns positive again (due to incoming transactions). The advantage of this approach is that the system remains live despite conflicting transactions. As a downside, a malicious party might be able to spend its entire balance k times. This problem appears inevitable in asynchronous networks, and additional assumptions might be necessary if we want to have a better " overspending bound."

The second, and probably the most straightforward approach, is to *rollback* any transaction tx appearing in a proof, i.e., removing tx and every transaction depending on tx from transaction histories. Surely, this comes with the downside of invalidating a previously accepted transaction, which might affect correct system members in real life. As an alternative way of compensating correct processes in this case, the application might opt to redistribute the stake from the misbehaving parties among the harmed members.

Given a strategy for the reconfiguration of system members and states, an interesting course to follow would be in self-reconfigurable systems [17]: the protocol automatically rearrange trust assumptions and merge conflicting histories to keep the system live.

Composition of trust. Alpos et al. [2] show how to compose trust models of different (possibly disjoint) systems. Given two asymmetric fail-prone systems $[\mathcal{F}_1, ..., \mathcal{F}_n]$ and $[\mathcal{F}'_1, ..., \mathcal{F}'_m]$ and matching decentralized quorum systems, a *composed* trust model can be constructed. In the context of our relaxed cryptocurrency protocols, it is appealing to understand how the spending number of a composition of two independent systems may depend on the spending number of its components.

References

1. Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011. doi:10.1145/1944345.1944348.
2. 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.
3. Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing*, pages 163–180. Springer, 2015. doi:10.1007/978-3-319-22846-4_10.
4. Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020. doi:10.1145/3419614.3423249.
5. João Paulo Bezerra, Petr Kuznetsov, and Alice Koroleva. Relaxed reliable broadcast for decentralized trust. In *International Conference on Networked Systems*, 2022.
6. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.
7. Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26, 1983. doi:10.1145/800221.806706.
8. Christian Cachin. Asymmetric distributed trust. In *International Conference on Distributed Computing and Networking 2021*, pages 3–3, 2021. doi:10.1145/3427796.3433933.
9. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011. doi:10.1007/978-3-642-15260-3.
10. Christian Cachin, Giuliano Losa, and Luca Zanolini. Quorum systems in permissionless networks. In *OPODIS*, 2022. doi:10.4230/LIPICS.OPODIS.2022.17.
11. Christian Cachin and Björn Tackmann. Asymmetric distributed trust. In *OPODIS*, volume 153, pages 7:1–7:16, 2019. doi:10.4230/LIPICS.OPODIS.2019.7.
12. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, feb 1999.
13. Brad Chase and Ethan MacBrough. Analysis of the xrp ledger consensus protocol. *arXiv preprint*, 2018. arXiv:1802.07242.
14. Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2019:587, 2019. URL: https://eprint.iacr.org/2019/587.
15. Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 26–38. IEEE, 2020. doi:10.1109/DSN48063.2020.00023.
16. Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 357–375. Springer, 2007. doi:10.1007/978-3-540-76900-2_22.
17. Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Accountability and reconfiguration: Self-healing lattice agreement. *CoRR*, abs/2105.04909, 2021. arXiv:2105.04909.
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. Álvaro García-Pérez and Alexey Gotsman. Federated byzantine quorum systems (extended version). *arXiv preprint*, 2018. arXiv:1811.03642.

20 Álvaro García-Pérez and Maria A Schett. Deconstructing stellar consensus (extended version). *arXiv preprint*, 2019. arXiv:1911.05145.

21 Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *DSN*, pages 568–580. IEEE, 2019. doi:10.1109/DSN.2019.00063.

22 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In Peter Robinson and Faith Ellen, editors, *PODC*, pages 307–316. ACM, 2019. doi:10.1145/3293611.3331589.

23 Saurabh Gupta. *A non-consensus based decentralized financial transaction processing model with support for efficient auditing*. Arizona State University, 2016.

24 Andreas Haeberlen and Petr Kuznetsov. The Fault Detection Problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS'09)*, dec 2009. doi:10.1007/978-3-642-10877-8_10.

25 Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. The case for byzantine fault detection. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep'06)*, nov 2006. URL: https://www.usenix.org/conference/hotdep-06/case-byzantine-fault-detection.

26 Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, oct 2007. doi:10.1145/1294261.1294279.

27 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006. doi:10.1007/11818175_27.

28 Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009. doi:10.1145/1658357.1658358.

29 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, volume 153 of *LIPIcs*, pages 31:1–31:17, 2019. doi:10.4230/LIPICS.OPODIS.2019.31.

30 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In Hagit Attiya, editor, *DISC*, volume 179 of *LIPIcs*, pages 27:1–27:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.DISC.2020.27.

31 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, jul 1978. doi:10.1145/359545.359563.

32 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. doi:10.4230/LIPICS.DISC.2019.27.

33 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998. doi:10.1007/S004460050050.

34 David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32, 2015.

35 R Miller. *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20 22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Springer Science & Business Media, 2013. doi:10.1007/978-1-4684-2001-2.

36 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

37 David Schwartz, Noah Youngs, Arthur Britto, et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8):151, 2014.

38 Isaac C. Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers. Heterogeneous paxos: Technical report. In *OPODIS*, 2020. doi:10.4230/LIPIcs.OPODIS.2020.5.

39 Robert Endre Tarjan and Anthony E Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977. doi:10.1137/0206038.

40 Marko Vukolić et al. The origin of quorum systems. *Bulletin of EATCS*, 2(101), 2013. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/183.

41 Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. White paper, 2015.

42 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019. doi:10.1145/3293611.3331591.

Bounds on Worst-Case Responsiveness for Agreement Algorithms

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Jennifer L. Welch
Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA

Abstract

We study the worst-case time complexity of solving two agreement problems, consensus and broadcast, in systems with n processes subject to no more than t process failures. In both problems, correct processes must decide on a common value; in the consensus problem, each process has an input and if the inputs of correct processes are all the same, then that must be the common decision, whereas in the broadcast problem, only one process (the sender) has an input and if the sender is correct, then its input must be the common decision. We focus on systems where there is an upper bound Δ on the message delivery time but it is expected that typically, messages arrive much faster, say within some time d. While Δ may or may not be known in advance, d is inherently unknown and specific to each execution. The goal is to design deterministic algorithms whose running times have minimal to no dependence on Δ, a property called *responsiveness*.

We present a generic algorithm transformation that, when applied to appropriate eventually-synchronous consensus (or broadcast) algorithms, results in consensus (or broadcast) algorithms for send omission failures, authenticated Byzantine failures, and unauthenticated Byzantine failures whose running times have no dependence on Δ; their worst-case time complexities are all $O(td)$, which is asymptotically optimal. The algorithm for send omission failures requires $n > 2t$, while those for Byzantine failures, both authenticated and unauthenticated, require $n > 3t$. The failure-resilience of the unauthenticated Byzantine algorithm is optimal.

For authenticated Byzantine failures, existing agreement algorithms provide worst-case time complexity $O(t\Delta)$ when n is at most $3t$. (When $n \leq 2t$, broadcast is solvable while consensus is not.) We prove a lower bound on the worst-case time complexity of $\lfloor (3t - n)/2 \rfloor \, d + \Delta$ when n is at most $3t$. Although lower bounds of Δ and $(t + 1)d$ were already known, our new lower bound indicates that, at least when $n \leq 2t$, it is impossible for an algorithm to pay these bounds in parallel.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases bounded-delay model, basic round model, omission failures, Byzantine failures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.32

Funding *Hagit Attiya*: partially supported by the Israel Science Foundation (grant 22/1425).

1 Introduction

Reaching agreement in the presence of faulty processes is a fundamental problem in distributed computing, with applications ranging from control systems and databases to cloud storage and blockchains. We consider two variations of reaching agreement. In the *consensus* problem, every process begins with an input and every correct process must reach the same decision; if all the correct processes have the same input, then that must be the common decision, a condition called *strong unanimity*. In the *broadcast* problem, there is a distinguished "sender" process that begins with an input and every correct process must reach the same decision; if the sender is correct, then the sender's input must be the common decision. The consensus and broadcast problems are closely related to each other, and in fact can be transformed to

each other in most situations very simply: Broadcast can be solved by having the sender send its input to all the processes and then having the processes run a consensus algorithm with each process' input being the value received from the sender. Consensus can be solved by using one copy of a broadcast algorithm for each process to disseminate its input and then having each process apply a common function to the vector of broadcast decisions to obtain the consensus decision.

We consider *send omission* failures, in which faulty processes fail to send some messages, and *Byzantine* (i.e., malicious) failures, in which faulty processes can change state arbitrarily and send messages with arbitrary content. Cryptographic developments allow the assumption that processes have access to *authentication* primitives to mitigate the effects of Byzantine failures. Note that if half or more of the processes are Byzantine, then the consensus problem, as stated above, is not solvable, and we only consider the broadcast problem. In this case, the transformations between consensus and broadcast discussed above work if we consider a weaker version of consensus in which the decision must be the common input only if all the processes are correct, a validity condition called *weak unanimity*. We assume a *permissioned* model, in which there is a known fixed set of n processes, up to t of which can be faulty.

Agreement problems have been studied under a variety of assumptions about the timing behavior of the system. Initially, the lock-step rounds model was considered (e.g., [25]), and then extended to more realistic synchronous models, with a known upper bound on message delay Δ. Relaxations of this model include when Δ is not known and when Δ only starts holding after some unknown *global stabilization time* (GST) [13]. In the asynchronous model, there is no upper bound on the message delays (cf. [16]).

We consider a *bounded-delay* system model, where message delay is at most Δ. Motivated by real-world networks in which the message delays in an execution are typically much smaller than the maximum possible delay Δ, researchers looked for algorithms whose performance is adaptive with respect to the maximum message delay actually experienced in each execution, call it d. Herzberg and Kutten [19] proposed this style of analysis and gave an algorithm to detect message forwarding faults with good performance under it. This behavior is dubbed *responsiveness* by Pass and Shi [23].

When Δ is known, classic agreement algorithms for the lock-step rounds model, with optimal $t + 1$ rounds (e.g., [12]), can be adapted by using Δ time to simulate each round, but this leads to algorithms with $\Theta(t\Delta)$ worst-case running times. In the other direction, the $t + 1$ round lower bound of [12] for agreement implies a lower bound of $\Omega(td)$ on the worst-case running time.

This paper investigates the possibility of avoiding or minimizing dependence on Δ in the worst-case running time for consensus and broadcast algorithms. In situations where d is much smaller than Δ, this would provide a significant speedup, allowing faster agreement on various decisions and actions.

Our first result (Theorem 2) is an algorithm transformation that takes any consensus or broadcast algorithm designed for a specific timing model called the *basic round model* [13] and produces an algorithm for the bounded-delay model with responsiveness that depends only on d. In the basic round model, processes take steps in lock-step rounds, messages are only received in the round in which they are sent, and after some unknown round number, called GST, no messages are lost. The transformed algorithm solves the same problem as the input algorithm, tolerates the same number and type of process failures, and has running time $O(Td)$, where T is the number of rounds after GST required for the original algorithm to decide. The transformed algorithm does not require that Δ be known, or that it even exist; that is, it could be that there is a bound d on the message delay in each particular execution, but no global bound over all executions.

Table 1 Time complexity lower bounds for several values of n and t, ignoring floors.

	$n = t + 2$	$n = \frac{4}{3}t$	$n = \frac{3}{2}t$	$n = 2t$	$n = 2t + 1$	$n = 3t - 1$	$n = 3t$
Thm. 9	$(t-1)d + \Delta$	$\frac{5t}{6}d + \Delta$	$\frac{3t}{4}d + \Delta$	$\frac{t}{2}d + \Delta$	$\frac{t-1}{2}d + \Delta$	$\frac{1}{2}d + \Delta$	Δ
[4]	$\frac{t}{2}\Delta$	3Δ	2Δ	Δ	N/A	N/A	N/A

The transformation is inspired by an algorithm of Dwork and Stockmeyer [14] for crash failures. We have generalized the approach so that it works for send omission and Byzantine failures, both with and without authentication. The main algorithmic idea is to run the simulated algorithm, periodically doubling an estimate of the message delay, until we eventually reach the actual message delay bound d, at which point the GST round is reached for the simulation.

By applying our transformation to relevant algorithms of Dwork, Lynch and Stockmeyer [13] for the basic round model, we obtain consensus (and broadcast) algorithms for send omission failures and Byzantine failures, both with and without authentication, that have asymptotically optimal worst-case running time of $O(td)$. To the best of our knowledge, these are the first algorithms for these types of failures with asymptotically optimal worst-case responsiveness.

The resulting algorithm for send omission failures requires $n > 2t$, while the other two require $n > 3t$. The $n > 3t$ resilience requirement is optimal for unauthenticated Byzantine failures [25]. For authenticated Byzantine failures, the resilience of $n > 3t$ needed for our algorithm is not optimal, as there are existing algorithms for consensus and broadcast that work with a larger fraction of faulty processes. In fact, the broadcast problem is solvable even when $n = t + 1$, as demonstrated by the algorithm of Dolev and Strong [12]. As mentioned earlier, consensus can be solved by using one copy of a broadcast algorithm for each process; however, if processes can be Byzantine when using this scheme, the strong unanimity validity condition requires a majority of correct processes, i.e., $n > 2t$. This requirement is inherent, as a simple partitioning argument (see [1]) shows that $n > 2t$ is a necessary condition for solving consensus with strong unanimity in the presence of Byzantine failures, even with authentication.

In an attempt to address the time complexity gap when $n \leq 3t$ in the case of authenticated Byzantine failures, we prove a lower bound (Theorem 9) when n is in the range $t + 2$ to $3t$ that the worst-case time complexity of consensus, even with weak unanimity must be at least $\lfloor (3t - n)/2 \rfloor \, d + \Delta$. Abraham, Nayak, Ren, and Xiang [4] showed a lower bound of $(\lfloor n/(n - t) \rfloor - 1)\Delta$ on the running time of broadcast for $n \leq 2t$. These two lower bounds are incomparable, depending on the relative values of t, d, and Δ; see Table 1. These lower bounds hint that the smaller n is compared to $3t$, the larger the reliance on the timeout must be. Ignoring constants, and looking at the key breakpoints, they indicate that when $n = t + 2$, the time complexity is $\Omega(t\Delta)$, which is asymptotically tight, and when $n = 2t + 1$, it is $\Omega(td + \Delta)$ compared with $\Theta(td)$ when $n = 3t + 1$.

Our focus on the *worst-case* responsiveness of broadcast and consensus algorithms, complements recent results for *optimistic* responsiveness, when the sender and a majority of the processes are correct, discussed in detail in the next section.

2 Related Work

Analyzing the running time of an algorithm for the bounded-delay model in terms of both d, the actual delay of messages, and Δ, the upper bound to time out message delivery, was proposed by Herzberg and Kutten [18, 19] in the context of message communication protocols. Subsequently, the idea was applied to consensus algorithms.

Table 2 Worst-case time complexity bounds for n processes with t crash failures.

	$t+1 < n \leq 2t$	$2t < n$
upper bound	$O(td+\Delta)$ [5, 6, 21]	$O(td)$ [14]
lower bound	$(2t-n)d+\Delta$ [14]	$(t+1)d$

Table 3 Worst-case time complexity bounds for n processes with t send omission failures.

	$t+1 < n \leq 2t$	$2t < n$
upper bound	$O(\min\{t/(n-t), \sqrt{\Delta/d}\}td+\Delta)$ [26]	$O(td)$ (Cor. 3)
lower bound	$(2t-n)d+\Delta$ (cf. Table 2)	$(t+1)d$

Attiya, Dwork, Lynch and Stockmeyer [6, 7] presented a *crash-tolerant* algorithm for consensus in the bounded-delay model with worst-case running time of $O(td+\Delta)$, for $n > t$. Constant-factor improvements were made in [5, 21]. Dwork and Stockmeyer [14] improved on this running time when $n > 2t$ with an algorithm whose worst-case time complexity is $O(td)$, which is asymptotically optimal; they also showed that when $n \leq 2t$ no algorithm can have worst-case time better than $(2t-n)d+\Delta$. Thus dependence on Δ, that is, requiring at least one time-out, is necessary if and only if $n \leq 2t$. See Table 2.[1]

Ponzio [26] extended the results of [6, 7] to *send omission failures* in the bounded-delay model, presenting a consensus algorithm for $n > t$ with worst-case time complexity $\Omega(td+\Delta)$. Bharali and Berman [9] further extended [26] to general omission failures as long as $n > 2t$, with the same asymptotic running time. Similar results are derived in a more structured manner by Attiya, Borran, Hutle, Milosevic and Schiper [5]. To the best of our knowledge, all previous algorithms for (send) omission failures in the bounded-delay model have worst-case running times that depend on Δ. In contrast, our algorithmic transformation, when applied to an appropriate base algorithm, yields an algorithm for send omission failures, where $n > 2t$, with worst-case time complexity $O(td)$, that is, with no dependence on Δ, which is asymptotically optimal. See Table 3.

For *authenticated Byzantine failures*, there is an algorithm for $n > 3t$ with worst-case running time of $O(d^2+t^2)$ [13]. It was designed for the partially synchronous model when Δ is unknown and thus it uses estimates of the observed actual message delays instead of Δ timeouts. It works in the bounded-delay model as well with no dependence on Δ, although the resilience is no longer optimal.[2] A recent paper of Civit et al. [11] presents a communication-optimal algorithm for the partially synchronous model in which Δ holds eventually (i.e., the eventually-synchronous model). It requires $n > 3t$ and has worst-case running time of $O(t\Delta)$ after Δ starts holding. This algorithm also works in the bounded-delay model and has worst-case running time of $O(t\Delta)$ and optimal communication complexity, although the resilience is no longer optimal. Our results on worst-case running time together with the upper and lower bounds that follow from the classical round-based results are summarized in Table 4.

[1] The lower bounds in all the tables implicitly include the maximum of the expression given and $(t+1)d$, which is due to the $(t+1)$-round lower bound when $n > t+1$ [12].

[2] Algorithm 2^2 in [13, Section 4.2]. The discussion after Theorem 4.2 states $O(N^2+\Delta^2)$, but N (number of processes) can be reduced to t by allowing multiple messages to be sent in each round, and Δ corresponds to our d as their algorithm is for the unknown-Δ model.

Table 4 Worst-case time complexity bounds for n processes with t authenticated Byzantine failures (only broadcast in the range $t+1 < n \leq 2t$).

	$t+1 < n \leq 2t$	$2t < n \leq 3t$	$3t < n$
upper bound	$(t+1)\Delta$	$(t+1)\Delta$	$O(td)$ (Cor. 4)
lower bound	$\max\{\lfloor(3t-n)/2\rfloor d + \Delta,$ $(\lfloor n/(n-t)\rfloor - 1)\Delta\}$ (Thm. 9 and [4])	$\lfloor(3t-n)/2\rfloor d + \Delta$ (Thm. 9)	$(t+1)d$

Table 5 Worst-case time complexity bounds for n processes with t (unauthenticated) Byzantine failures.

	$n \leq 3t$	$3t < n$
upper bound	impossible [15, 25]	$O(td)$ (Cor. 5)
lower bound	impossible [15, 25]	$(t+1)d$

For *Byzantine failures without authentication*, $n > 3t$ is a necessary condition even in lock-step synchronous systems [15, 25]. To handle such failures, we can apply our transformation to the algorithm in Section 3.2.3 of Dwork, Lynch and Stockmeyer [13] to obtain a consensus algorithm with $O(t \cdot d)$ worst-case time complexity, which is asymptotically optimal. See Table 5.

Recently, the broadcast problem in the presence of authenticated Byzantine failures has captured attention, motivated by state machine replication[3] and blockchains. In a complementary direction to the results in this paper which are for *worst-case* running time, the primary focus is on *good-case* behavior, where "good-case" means the sender is correct, as well as behavior in even more optimistic situations where, in addition to the sender, a large majority or even all the processes are correct. Typically, the worst-case running times of these algorithms, including when the sender is faulty, are either not considered or shown to be $\Omega(t\Delta)$.

Broadcast algorithms with good-case time complexity of $O(d)$ have been proposed by Castro and Liskov (PBFT) [10], Pass and Shi (Hybrid consensus) [23], Yin, Malkhi, Reiter, Gueta and Abraham (HotStuff) [30], and Abraham, Nayak, Ren, and Xiang [4]. All these algorithms assume $n > 3t$. However, there is a fundamental barrier to achieving such good performance if the fraction of faulty processes is larger: Abraham, Malkhi, Nayak, Ren and Yin [2] prove that if $n \leq 3t$, then the good-case running time, i.e., when the sender is correct, must be at least Δ; their proof can be viewed as a quantitative version of the proof of Theorem 4.4 in [13] that consensus is impossible with $n < 3t$ for authenticated Byzantine failures when the upper bound on message delay is either unknown or only holds eventually. Pass and Shi [23] have an analogous result for the permissionless model.

When n is between $2t+1$ and $3t$, several algorithms have been proposed that have $O(d)$ running time under some optimistic conditions, but by necessity they pay a price of at least Δ in the running times in other good-case executions. Pass and Shi's Thunderella [24] has $O(d)$ time when a super-majority of the processes are correct but otherwise has time $O(t\Delta)$. Follow-up work has provided algorithms that have $O(d)$ running time under certain optimistic conditions and $O(\Delta)+O(d)$ running time under other optimistic conditions (e.g., [2–4, 22, 27]). Tradeoff lower bounds on the running times achievable under different optimistic conditions have been proved [22, 27].

[3] State machine replication can be viewed as a collection of repeated instances of broadcast by various senders, where processes must agree on the values sent by the senders as well as an ordering for the broadcasts.

If n is between $t+1$ and $2t$, an algorithm with good-case complexity of $O(\frac{n}{n-t}\Delta)$ has been proposed by Abraham, Nayak, Ren and Xiang [4] based on an algorithm of Wan, Xiao, Shi and Devadas [29]. As discussed in the introduction, there is an asymptotically matching lower bound of $\left(\left\lfloor\frac{n}{n-t}\right\rfloor - 1\right)\Delta$ in [4], which was inspired by a proof in [17].

We are not considering randomized algorithms for asynchronous systems, since their worst-case running times would be infinite, due to the impossibility of solving fault-tolerant consensus and broadcast deterministically in asynchronous systems [16]. It should be noted, however, that they have responsive running times, depending on d and not on Δ, *in expectation*, since they are inherently asynchronous.

3 Preliminaries

3.1 The Bounded-Delay System Model

Fix positive integers n, t, and Δ. We present a model of an algorithm for n processes, t of which may be faulty, that communicate by sending messages over reliable, point-to-point channels with delay at most Δ.

There is a set M of messages. Each *message* is a triple (s, m, r), where s indicates the sending process, m is the message contents, and r indicates the receiving process.

There is a set of n processes, $p_1, p_2, \ldots, p_n$, where each *process* p_i is a state machine with a (possibly infinite) set of states Q_i and two transition functions. The set of states Q_i includes a subset of *initial states*. The two transition functions of p_i are $\delta_i^m : Q_i \to 2^M$ (which produces the next set of messages to send depending on the current state) and $\delta_i^s : Q_i \times 2^M \to Q_i$ (which produces the next state depending on the old state and set of messages received).

A *history* of process p_i is an infinite sequence $(M_0^R, q_0, M_0^S, M_1^R, q_1, M_1^S, \ldots)$, where $q_t \in Q_i$, $M_t^R \subseteq M$, and $M_t^S \subseteq M$ for all $t \geq 0$. The triple (M_t^R, q_t, M_t^S) is the *step* of p_i occurring at *time* t, in which p_i receives the set M_t^R of messages, changes its local state to q_t, and sends the set M_t^S of messages. A history is *correct* if q_0 is an initial state of p_i, $q_t = \delta_i^s(q_{t-1}, M_t^R)$ for all $t \geq 1$, and $M_t^S = \delta_i^m(q_t)$ for all $t \geq 1$. In other words, the next state and the messages sent are determined by p_i's transition functions.

An *execution* α is a set of n histories, one for each process, satisfying the following properties.

- At least $n - t$ of the histories are correct; the corresponding processes are the *correct processes* while the rest are *faulty*.
- Every message in an M_t^S (resp., M_t^R) component of p_i's history has i as its sender (resp., recipient). That is, neither the sender nor the channel can lie about the sender, and the channel does not misdeliver messages.
- There exists a bijection from the set of messages appearing in the M_t^S components of the histories to the set of messages appearing in the M_t^R components of the histories such that every message sent is received exactly once and every message received is sent. A message contained in the M_b^R-th component of a process history (the recipient) and contained in the M_a^S-th component of a process history (the sender) is defined to have *delay* $b - a$.
- There exists $d(\alpha) \leq \Delta$ such that the delay of every message is at least 1 and at most $d(\alpha)$; the lower bound of 1 implies that M_0^R is the empty set in every history. When α is understood from context, we denote the bound simply as d.

Since every process takes a step at every nonnegative integer time in its history, in an execution all processes take steps together at every nonnegative integer time.

To model *send omission* failures, we restrict the faulty processes to behave the same as the correct processes, except that in each step, M_t^S can be a subset of $\delta_i^m(q_t)$, reflecting the fact that some prescribed messages are not sent. For *Byzantine* failures, the faulty processes need not follow their transition functions when changing state or sending messages. When an *authentication* mechanism is available, the possible behaviors of the faulty processes are limited: each message sent by a process is signed by the process and no signature can be forged by another process. Thus if process p_i sends a message to process p_j claiming that process p_k sent message m, then p_k did indeed send m. Such a mechanism can be implemented, for instance, using public-key cryptography.

3.2 Definition of Agreement Problems

Every execution of an algorithm solving the *consensus* problem must satisfy the following conditions. Every process p_i starts with an input value from some finite domain of values V. This is modeled by having $|V|$ initial states in the state set Q_i of p_i, one for each possible input $v \in V$.

Termination: Every correct process must eventually make an irrevocable decision on a value from V. We model this by requiring $|V|$ nonintersecting nonempty subsets of Q_i, one for each decision $v \in V$, and requiring the transition function δ_i^s to stay inside each subset. (Processes continue running even after they have decided.)

Agreement: The decision values of all correct processes must be the same.

We also require one of the following validity conditions, which relates the common decision value to the input values.

Strong Unanimity: If the input values of all the processes are the same, say v, then every correct process decides v.

Weak Unanimity: If the input values of all the processes are the same, say v, and there are no failures, then every process decides v.

For both validity conditions, if the inputs are different, then the common decision can be any input value.

In the broadcast problem, a single process p_s is identified as a *sender*. In every execution of an algorithm solving the *broadcast* problem, the sender starts with an input m from some finite domain of values V (modeled as for the consensus problem), and the following conditions must be satisfied:

Termination: Every correct process must eventually make an irrevocable decision on a value from V (modeled as for the consensus problem).

Agreement: All correct processes must decide on the same input m'.

Validity: If the sender p_s is correct then $m' = m$.

As mentioned in the introduction, the broadcast problem can be solved by having the sender send its input to all the processes and then having the processes run a consensus algorithm with each process' input being the value received from the sender. Thus consensus with weak unanimity is equivalent to broadcast and is not subject to the $n > 2t$ requirement for strong unanimity when faulty processes are Byzantine.

We are interested in the *worst-case running time* of agreement algorithms, defined as follows. For execution α, the running time is the smallest time t such that every correct process has decided by time t. The worst-case running time of an algorithm is the maximum, over all executions α, of the running time of α.

4 Upper Bounds on Worst-Case Running Times

In this section, we present algorithms for consensus in the bounded-delay model for send omission, authenticated and unauthenticated Byzantine failures that have asymptotically optimal worst-case running time of $O(fd)$. These consensus algorithms can be converted into algorithms for broadcast with no increase in the (asymptotic) running time using the method mentioned in Section 3. As mentioned before, if Δ is known, then any consensus algorithm A for the lock-step rounds model can be adapted to work in the bounded-delay model by simulating each round of A using Δ time. The resulting algorithm inherits the same fault-tolerance (type of failures and relationship between n and t) and the same unanimity condition as A. However, the running time becomes $T \cdot \Delta$, where T is the round complexity of A after GST, which must be at least $t + 1$ (e.g., [12, 20]).

The main result of this section shows how to significantly improve on the running time of $(t + 1) \cdot \Delta$ when $n > 3t$, by replacing the dependence on Δ with dependence on the per-execution delay bound d, without relying on any knowledge of Δ. The algorithms result from applying a new transformation to appropriate base algorithms. In Section 4.1, we present the transformation. In Sections 4.2, 4.3, and 4.4, we explain how to use the transformation to achieve asymptotically time-optimal algorithms for send omission failures, authenticated Byzantine failures, and unauthenticated Byzantine failures, respectively.

4.1 Transformation from Basic Round Model to Bounded-Delay Model

We present a generic transformation inspired by an algorithm for crash failures in [14]. The transformation operates on algorithms designed for a more abstract model, called the *basic round model* [13].

▶ **Definition 1** (Basic Round Model)**.** *In the* basic round model*, processes operate in lock-step rounds. In each round, every correct process sends a set of messages to the other processes, then receives a subset of the messages sent at that round destined for it, and then performs some local computation. Starting at some unknown round, called* GST*, every message sent by a correct process to a correct process in a round is received in that round.*

Suppose A is an algorithm for the basic round model that solves consensus for n processes in the presence of up to t process failures, and decides by T rounds after GST. We show how to transform A into algorithm $Tr(A)$ that solves consensus for the same n and t and the same type of process failure in the bounded-delay model. The key feature of the transformed algorithm is that its running time in any execution is $O(T \cdot d)$, where d is the actual maximum message delay in the execution. The transformed algorithm does not rely on knowledge of Δ.

In the transformation, each process partitions its steps into groups so that the g-th group contains $2^g \cdot T$ steps, for $g = 1, 2, 3 \ldots$. Group g simulates T rounds of A using 2^g as an estimate of Δ in this group; thus each simulated round in the group takes 2^g steps.[4] In more detail, group g consists of rounds $g \cdot T + 1$ through $(g + 1) \cdot T$. In the first step of round r, which is part of group g, a process sends the messages it is supposed to send in round r of A, tagged with the round number. It then waits 2^g steps and collects all messages received that are tagged with round number r; any other messages are discarded. It then uses the messages received and its current simulated local state to simulate its round r state transition according to A, producing the new local state and set of messages to send at the beginning of the next simulated round. Pseudocode for the transformation is presented in Algorithm 1.

[4] To avoid the corner case when a round is simulated by a single step, we start with $g = 1$ instead of $g = 0$.

Algorithm 1 Transformation of algorithm A for the basic round model that decides by T rounds after GST into algorithm $Tr(A)$ for the bounded-delay model; code for process p_i.

initially:

1: $g := 1$ ▷ *group number:* $1, 2, 3, \ldots$
2: $r := 1$ ▷ *round number:* $1, 2, 3, \ldots$
3: $s := 1$ ▷ *counts steps in current round:* $1, 2, \ldots, 2^g, 1, 2, \ldots$
4: $sim\_state :=$ initial state of p_i in A
5: $M_R := \emptyset$ ▷ *set of messages received during current round of* A
6: $M_S :=$ set of messages to send at beginning of round 1 of A

7: **while** true **do** ▷ *execute a step*
8: add to M_R all messages received at this step with tag r ▷ *part of round* r *receive*
9: **if** $s = 1$ **then** ▷ *first step of round* r
10: send M_S (each message is tagged with r) ▷ *round* r *send*
11: **else if** $s = 2^g$ **then** ▷ *last step of round* r
12: $(M_S, state) := \delta_i^A(sim\_state, M_R)$ ▷ *round* r *computation*
13: $sim\_state := state$ ▷ *update simulated state;* M_S *will be sent in next step*
14: **if** $r = (g + 1) * T$ **then** ▷ *end of group* g
15: $g := g + 1$ ▷ *start next group*
16: $r := r + 1$ ▷ *start next round of* A
17: $M_R := \emptyset$ ▷ *clear set of received messages for new round*
18: $s := 1$ ▷ *reset step counter for new round*
19: **else** ▷ *neither first nor last step of round* r
20: $s := s + 1$

▶ **Theorem 2.** *Let A be a consensus algorithm for n processes in the basic round model that tolerates up to t process failures and decides by T rounds after GST. Then $Tr(A)$, the result of applying Algorithm 1 to A, is a consensus algorithm with the same unanimity condition for n processes in the bounded delay model that tolerates up to t process failures of the same type and has running time $O(T \cdot d)$ in every execution with maximum message delay d.*

Proof. Consider any execution τ of $Tr(A)$ with upper bound d on the message delays. We can extract from it an execution α of A in the basic round model as follows. For every $r \geq 1$ and every process p_i, the send of round r by p_i in α corresponds to the execution by p_i in τ of Line 10 when p_i's local variable r_i equals r. The receive of round r by p_i in α is distributed over all the executions by p_i in τ of Line 8 when p_i's local variable r_i equals r. The compute of round r in α corresponds to the execution in τ of Line 12 when p_i's local variable r_i equals r. Once $2^g \geq d$, that group (g) of steps will consist of a simulation of T rounds of A after GST in the basic round model, since every message sent in a simulated round that is part of group g (or later) is received in the same simulated round. Since A is correct, α satisfies termination, agreement, and its designated unanimity condition. Thus the same is true of τ and $Tr(A)$ is correct.

We now calculate the time complexity in execution τ of the transformed algorithm. By the assumption that A decides by T rounds after GST in the basic round model and the correspondence between τ and α just presented, correct processes in τ decide by the end of group g, where g is the smallest integer such that $2^g \geq d$. The time until group g ends is at most

$$T \cdot \sum_{g=0}^{\lceil \log d \rceil} 2^g = T \cdot (2^{\lceil \log d \rceil} - 1) = \Theta(T \cdot d). \quad \blacktriangleleft$$

4.2 Transformed Algorithm for Send Omission Failures

For the base algorithm of our transformation for send omission failures, we start with the algorithm in [13, Section 3.2.1] for consensus in the basic round model. We modify the algorithm to include the optimizations discussed in Remark 1 (p. 302) of [13] that reduce the running time in the basic round model to be $O(t)$ rounds after GST. Call the modified algorithm A_{om}. The algorithm works for n processes, t of which can experience send omission failures, and achieves strong unanimity assuming $n \geq 2t + 1$. For completeness, we present a brief overview of A_{om}, followed by detailed pseudocode[5], and key ideas for correctness (see [13] for detailed correctness proof and worst-case time complexity analysis).

A_{om} is structured as a series of phases, each consisting of four rounds; process $p_{k \bmod n}$ is the *leader* of phase k. Once a correct process is the leader after GST, processes quickly decide. Each process keeps track of the set of values that it has learned are inputs (cf. *Proper* variable); this information is propagated on all messages sent. It also keeps track of the set of values that are candidates for decision (cf. *Locked* variable); each value is associated with the phase number in which it is chosen.

In the first round of a phase, each process sends to the leader a LIST message with all values in its *Proper* set that have no competing value in its *Locked* set. In the second round, the leader chooses any one of the values received in the first round (if any) as its preference, and sends a LOCK message with that value to all the processes. In the third round, if a process receives the LOCK message from the leader, then it updates its *Locked* set and sends an acknowledgment (ACK message) back to the leader. If the leader receives at least $n - t$ acknowledgements, then it decides on its preference. In the fourth round, the leader sends its *Locked* set and its decision (in a LOCK_REL_DEC message) to all the processes. Each recipient removes from its own *Locked* set any value that has a smaller associated phase than any locked value received from the leader. If the recipient has not yet decided, then it adopts the decision in the leader's message.

The pseudocode for A_{om} appears in Algorithm 2; it assumes $n > 2f$. In the pseudocode, *leader*(k) is $p_{k \bmod n}$ and "update (v, k) in *Locked*" means to add (v, k) to the local set variable *Locked* and delete any (v, k') in *Locked* where $k' < k$.

We now sketch the key arguments for correctness. First, it is straightforward to argue that only input values of some (nonfaulty) processes are added to the *Proper* set. Since only values in this set are ever proposed or locked, it follows that if the initial value of every process is v, then no value $v' \neq v$ is ever proposed or locked. This implies *validity*.

The key property that is proved is *lock continuation*, namely, if a process decides on v at the end of phase k, then for every for every phase $k' \geq k$, at least $f + 1$ processes hold a lock (v, k'). The lock continuation property implies that no value $v' \neq v$ is proposed or decided in later phases, which ensures *agreement*.

Finally, it is also straightforward to prove that at most one value v is proposed in the second round of each phase k, and hence, there is a lock only on (v, k). Then, consider what happens at the first phase k after GST such that the leader $p_{k \bmod n}$ is nonfaulty. At phase k, the leader $p_{k \bmod n}$ proposes a single value and since the communication is reliable after GST, it will get acknowledgements for its proposed value and decide. Furthermore, all other processes will receive the decision value in the last round of the phase, and they will decide as well. This implies *termination* within t phases, and hence, $O(t)$ rounds, after GST.

[5] This algorithm, like those used in the next two subsections, is only presented in prose in [13]; we believe we have correctly captured the intent.

Algorithm 2 Consensus in the basic round model, for send omission failures; code for process p_i, $0 \leq i < n$.

initially:
1: $Proper := \{input\}$ ▷ *set of values known to be inputs; input is p_i's input*
2: $Locked := \emptyset$ ▷ *set of locked values with phase numbers*
3: $decision := \perp$ ▷ *p_i's decision*
4: $M := \emptyset$ ▷ *set of messages received most recently*

5: **procedure** calc_proper(M):
6: add $Proper$ set in m to (local variable) $Proper$ for each $m \in M$

7: **first round of phase** k **($k = 1, 2, \ldots$):** ▷ *all-to-leader*
8: $PA := \{v \mid v \in Proper$ and no $(u, *)$ is in $Locked$ with $u \neq v\}$ ▷ *proper and acceptable*
9: send $\langle$LIST, k, PA, $Proper\rangle$ to $leader(k)$
10: receive set M of messages; call calc_proper(M)

11: **second round of phase** k**:** ▷ *leader-to-all*
12: **if** $p_i = leader(k)$ **then** ▷ *choose value to propose*
13: $W :=$ union of PA set in m for each $m \in M$ ▷ LIST *messages from previous round*
14: **if** $|W| > 0$ **then**
15: $pref :=$ arbitrary element of W
16: send $\langle$LOCK, k, $pref$, $Proper\rangle$ to all
17: receive set M of messages; call calc_proper(M)

18: **third round of phase** k**:** ▷ *all-to-leader*
19: **if** $M = \{m\}$ for some message m **then** ▷ *M is either empty or one* LOCK *message*
20: update (v, k) in $Locked$ where v is $pref$ in m
21: send $\langle$ACK, k, $Proper\rangle$ to $leader(k)$
22: receive set M of messages; call calc_proper(M) ▷ *only $leader(k)$ has $|M| > t$*
23: **if** $(|M| \geq n - t)$ and $(decision = \perp)$ **then** ▷ *got $n - t$* ACK*'s and undecided*
24: $decision := pref$ ▷ *p_i decides when it is $leader(k)$*

25: **fourth round of phase** k**:** ▷ *leader-to-all*
26: **if** $p_i = leader(k)$ **then**
27: send $\langle$LOCK_REL_DEC, k, $Locked$, $decision$, $Proper\rangle$ to all
28: receive set M of messages; call calc_proper(M)
29: **for** each $\langle$LOCK_REL_DEC,$k, L, d, *\rangle$ message in M **do**
30: **for** each $(w, h') \in L$ **do** ▷ *release old locks*
31: delete all (v, h) in $Locked$ with $(w \neq v)$ and $(h' \geq h)$
32: **if** $(d \neq \perp)$ and $(decision = \perp)$ **then**
33: $decision := d$ ▷ *p_i decides, might not be $leader(k)$*

After applying our transformation to A_{om} (Algorithm 2), we obtain an asymptotically time-optimal algorithm for the bounded-delay model:

▶ **Corollary 3.** *There exists a consensus algorithm (with strong unanimity) for n processes that tolerates up to t send omission failures, $n \geq 2t + 1$, and has running time $O(t \cdot d)$ in every execution with maximum message delay d.*

4.3 Transformed Algorithm for Authenticated Byzantine Failures

For the base algorithm of our transformation for authenticated Byzantine failures, we use the algorithm in [13, Section 3.2.2] with modifications to decide within $O(t)$ rounds after GST in the basic round model. Let A_{au} be the modified algorithm. The algorithm works for n processes, t of which can be Byzantine, and uses authentication to achieve strong unanimity, as long as $n \geq 3t + 1$. For completeness, we present a brief overview of A_{au}, followed by detailed pseudocode, and key ideas for correctness (see [13] for detailed correctness proof and worst-case time complexity analysis).

A_{au} is structured similarly to A_{om}, with four rounds per phase. We next mention the key differences. (1) Messages are signed before being sent. (2) Keeping track of the input values of correct processes is more involved due to the possibility that faulty process lie, requiring that processes tag every message sent with its input as well as its *Proper* set. If no input occurs at least $t + 1$ times once $2t + 1$ inputs have been reported, then all values in V are possible decisions and are added to *Proper*; otherwise if a value appears in $t + 1$ *Proper* sets reported by other processes, then w is added *Proper*. See procedure calc_proper. (3) The preference chosen by the leader in the second round must have a "proof", meaning that it is contained in the values of at least $n - t$ LIST messages received in the first round. This proof (set of $n - t$ signed messages) is included in the leader's LOCK message. (4) In the third round, the leader decides on its preference if it receives at least $2t + 1$ acknowledgements. (5) In the fourth round, locks are sent all-to-all, not just from the leader. (6) In order for a process to decide in the fourth round when it is not the leader, it must receive at least $t + 1$ identical decision values from other processes, not just one.

The pseudocode for A_{au} appears in Algorithm 3; it assumes $n > 3f$. Whenever a process receives a signed message, it handles the message only after validating the signature, and similarly for signed messages forwarded inside other messages. For clarity, we omit these checks from the code, and implicitly assume they are carried out. In the pseudocode, *leader*(k) is $p_{k \bmod n}$ and "update $E_i(v, k, proof)$ in *Locked*" means to add $E_i(v, k, proof)$ to the local set variable *Locked* and delete any $E_i(v, k', proof')$ in *Locked* where $k' < k$.

Much of the correctness proof for A_{au} is similar to that for A_{om}. A key difference is that the impossibility of two different values acquiring a valid lock in the same phase is now due to the fact that processes send proofs in their lock messages. Termination is also argued differently and holds because of the more involved management of proper values.

After applying our transformation to A_{au} (Algorithm 3), we obtain an asymptotically time-optimal algorithm:

▶ **Corollary 4.** *There exists a consensus algorithm (with strong unanimity) for n processes that tolerates up to t authenticated Byzantine failures, $n \geq 3t + 1$, and has running time $O(t \cdot d)$ in every execution with maximum message delay d.*

4.4 Transformed Algorithm for Unauthenticated Byzantine Failures

For the base algorithm of our transformation for unauthenticated Byzantine failures, we use the algorithm in [13, Section 3.2.3] with modifications to decide within $O(t)$ rounds after GST in the basic round model. Let A_B be the modified algorithm. The algorithm works for n processes, t of which can be Byzantine, and achieves strong unanimity, as long as $n \geq 3t + 1$. For completeness, we present a brief overview of A_B, followed by detailed pseudocode, and key ideas for correctness (see [13] for detailed correctness proof and worst-case time complexity analysis).

Algorithm 3 Consensus in the basic round model, for authenticated Byzantine failures; code for process p_i, $0 \leq i < n$.

initially:
1: $Proper := \{input\}$ ▷ *set of values known to be inputs; input is p_i's input*
2: $Input\_vals := \emptyset$ ▷ *keep track of input values reported*
3: $Other\_proper[]$:= array of sets of values, one per process; initially all $\emptyset$ ▷ *keep track of Proper sets reported*
4: $Locked := \emptyset$ ▷ *set of locked values with phase numbers and proofs*
5: $decision := \perp$ ▷ *p_i's decision*
6: $Other\_decisions := \emptyset$ ▷ *keep track of decisions reported*
7: $M := \emptyset$ ▷ *set of messages received most recently*

8: **first round of phase** k **(**$k = 1, 2, \ldots$**):** ▷ *all-to-leader*
9: $PA := \{v \mid v \in Proper$ and no $(u, *, *)$ is in $Locked$ with $u \neq v\}$ ▷ *proper and acceptable*
10: send $E_i\langle$LIST, k, PA, $Proper$, $input\rangle$ to $leader(k)$ ▷ *E_i is p_i's authentication function*
11: receive set M of messages; call calc_proper(M) ▷ *calc_proper definition appears below*

12: **second round of phase** k**:** ▷ *leader-to-all*
13: **if** $p_i = leader(k)$ **then**
14: $W := \emptyset$ ▷ *choose value to propose*
15: $Proof[v] := \emptyset$ for each $v \in V$
16: **for** each $m = E_j\langle$LIST, k, S, $*$, $*\rangle$ message in M **do** ▷ *from previous round*
17: **for** each $v \in S$ **do**
18: add m to $Proof[v]$
19: add v to W for each $v \in V$ such that $|Proof[v]| \geq n - t$
20: **if** $|W| > 0$ **then**
21: $pref$:= arbitrary element of W
22: send $E_i\langle$LOCK, k, $pref$, $Proof[pref]$, $Proper$, $inpu\rangle$ to all
23: receive set M of messages; call calc_proper(M)

24: **third round of phase** k**:** ▷ *all-to-leader*
25: **if** $M = \{m\}$ for some message m **then** ▷ *M is either empty or one* LOCK *message*
26: let v be the $pref$ in m
27: **if** $|proof$ in $m| \geq n - t$ **then**
▷ *at least $n - t$ processes find v proper and acceptable in this phase*
28: update $E_i(v, k, proof)$ in $Locked$
29: send $\langle$ACK, k, $Proper$, $input\rangle$ to $leader(k)$
30: receive set M of messages; call calc_proper(M) ▷ *only $leader(k)$ has $|M| > t$*
31: **if** $(|M| \geq 2t + 1)$ and $(decision = \perp)$ **then** ▷ *got $2t + 1$* ACKs *and undecided*
32: $decision := pref$ ▷ *p_i decides when it is $leader(k)$*
▷ *continued.....*

▷ .....*Continuation of Algorithm 3*

33: **fourth round of phase** k**:** ▷ *all-to-all*
34: send $\langle$LOCK_REL_DEC, k, *Locked*, *decision*, *Proper*, *input*$\rangle$ to all
35: receive set M of messages; call calc_proper(M)
36: **for** each $\langle$LOCK_REL_DEC, k, L, d, $*$, $*\rangle$ message in M **do**
37: **for** each $(w, h', proof')$ in L **do** ▷ *release old locks*
38: delete all $E_i(v, h, proof)$ in *Locked* with $(w \neq v)$ and $(h' \geq h)$
39: **if** there is no element in *Other_decisions* for j, where p_j is the sender of m **then**
40: add (j, d) to *Other_decisions*
41: **if** (there exist $t+1$ elements in *Other_decisions* with the same value, v) and (*decision* $= \perp$) **then**
42: *decision* $:= v$ ▷ p_i *decides, might not be leader*(k)

43: **procedure** calc_proper(M):
44: **for** each $m \in M$ **do**
45: let p_j be the sender of m, v be the input value in m, and pr be the *Proper* set in m
46: **if** there is no element in *Input_vals* for j **then**
47: add (j, v) to *Input_vals*
48: *Other_proper*$[j] := pr$ ▷ *over-write any previously reported Proper set from* j
49: **if** ($|$*Input_vals*$| = 2t+1$) and (no value occurs at least $t+1$ times in *Input_vals*) **then**
50: *Proper* $:= V$ ▷ *all possible inputs are valid*
51: **else**
52: **for** each $w \in V$ **do** ▷ *check if* w *is considered proper by* $t+1$ *processes*
53: **if** w appears in (at least) $t+1$ entries of *Other_proper*[] **then**
54: add w to *Proper*

To handle Byzantine failures, this algorithm replaces the authentication mechanism of A_{au} by having processes communicate using a reliable broadcast primitive. The high-level structure is similar to A_{om} and A_{au} except that each round of a phase is replaced with two rounds, that implement the reliable broadcast primitive. These pairs of rounds are called *superrounds*. The procedure of releasing locks, which was accomplished during the fourth round in each phase in the previous two algorithms, no longer requires additional communication and thus the lock release is done at the end of the third superround instead of during a fourth superround. As a result, each phase takes three superrounds, which is six rounds. The *reliable broadcast* primitive ensures three properties:

Correctness: If a correct process p broadcasts message m in a superround that starts after GST, then m is delivered from p at every correct process in the same superround.

Unforgeability: If a correct process p does not broadcast message m, then m is never delivered from p at any correct process.

Relay: If message m is delivered from p at any correct process in superround r, then m is delivered from p at every correct process by superround $r+1$ or GST, whichever is later.

Pseudocode for an implementation of the reliable broadcast primitive with a cost of two rounds for broadcast-deliver is given in [13], based on [28]. In the first round, the message to be broadcast is sent to all. In the second round, messages are echoed (sent again to all) and if at least $n-t$ distinct echoes of a message are received, that message is delivered. The complication is that, in order to handle the possible loss of messages before GST, messages continue to be echoed in all later rounds: if a process receives at least $n-2t$ distinct echoes of the same message, then it sends send an echo to all, and once it has received at least $n-t$ distinct echoes for a message it is delivered.

The pseudocode for A_B appears in Algorithm 4; it assumes $n > 3f$. We assume some mechanism for dropping messages that are ill-formed, duplicated, etc. from faulty processes.

After applying our transformation to A_B (Algorithm 4), we obtain an asymptotically time-optimal algorithm:

▶ **Corollary 5.** *There exists a consensus algorithm (with strong unanimity) for n processes that tolerates up to t Byzantine failures, $n \geq 3t + 1$, and has running time $O(t \cdot d)$ in every execution with maximum message delay d.*

5 Lower Bound on Worst-Case Running Time

In this section, we present a lower bound on the worst-case running time of any consensus or broadcast algorithm for n processes that tolerates t Byzantine failures and has access to an authentication mechanism. The lower bound is shown for consensus with weak unanimity, and thus it also holds for consensus with strong unanimity (when $n > 2t$) and for broadcast. Recall that Δ is the global upper bound on message delays across all executions and $d(\alpha)$ is the upper bound on message delays in a specific execution α. Our lower bound assumes that Δ is known, meaning that the algorithm can explicitly use Δ, e.g., to time out waiting for messages. On the other hand d is not known, as it can vary from execution to execution.

Recall that, as mentioned in [6], a simple adaptation of the $(t+1)$-round lower bound for crash failures in the synchronous model provides a lower bound of $(t+1)d(\alpha)$ time for every execution α. (Below, we use d instead of $d(\alpha)$, when α is clear from the context.) This lower bound shows that the algorithm of Corollary 4 has asymptotically optimal running time.

Our lower bound states that when $n \leq 3t$, the worst-case running time must be at least $\lfloor (3t-n)/2 \rfloor \cdot d + \Delta$. Table 1 displays our lower bound as well as the lower bound of $\lfloor n/(n-f) - 1 \rfloor \cdot \Delta$ when n is between $t+2$ and $2t$ [4].

Our result is inspired by one in [14] for crash failures, which showed a lower bound of $(2t-n)d + \Delta$ when $n \leq 2t$. Of course this bound also holds for authenticated Byzantine failures, but by exploiting the worse behavior of the faulty processes, we are able to increase the factor of $(2t-n)$ to $\lfloor (3t-n)/2 \rfloor$ and to increase the range to $n \leq 3t$. The main technical novelties are (1) finding the right partitioning of the processes so that the faulty processes can be substituted for (temporarily) disconnected correct processes, and (2) identifying the desired behavior of the faulty processes, and showing it is possible despite the use of authentication, which requires an involved argument.

The proof considers executions that mimic the behavior of the synchronous rounds model with crash failures. We call such executions "synchronized" and define them next.

Given a positive integer d and a history of process p_i, the subsequence $(M_t^S, M_{t+1}^R, q_{t+1}, \ldots, M_{t+d-1}^S, M_{t+d}^R, q_{t+d})$, where $t = (r-1)d$, is called *round r of p_i*, for $r \geq 1$. Given an execution, the collection of round r subsequences of all the processes is called *round r* (of the execution). Note that round r starts with the sending of messages at time $(r-1)d$ and ends with the receiving of messages and subsequent state changes at time rd, but does not including the sending of messages at time rd.

An execution is *synchronized* if

1. every message sent in round r is received in round r, $r \geq 1$, implying that $d(\alpha) \leq d$;
2. the behavior of a faulty process p_i only deviates from that of a correct process by sending a proper subset of the specified messages at some time t, i.e., $M_t^S \subsetneq \delta_i^m(q_t)$, and no messages subsequently; we say the process *fails in round r* if t is in round r; and
3. at most one process fails in each round.

An *s-round synchronized execution prefix* is the result of taking a synchronized execution and truncating each process history after the end of round s.

Algorithm 4 Consensus in the basic round model, for unauthenticated Byzantine failures; code for process p_i, $0 \leq i < n$.

initially:

Proper := {*input*} ▷ *set of values known to be inputs; input is p_i's input*
Input_vals := ∅ ▷ *keep track of input values reported*
Other_proper[] := array of sets of values, one per process; initially all ∅ ▷ *keep track of Proper sets reported*
Locked := ∅ ▷ *set of locked values with phase numbers and proofs*
decision := ⊥ ▷ *p_i's decision*
$M := \emptyset$ ▷ *set of messages delivered most recently*
$H := \emptyset$ ▷ *set of messages delivered so far*

procedure check_dec(H): ▷ *to speed up decision*
if H contains at least $t+1$ messages with different senders and the same non-⊥ decision value, say v) and (*decision* = ⊥) **then**
    decision := v

first superround of phase k **($k = 1, 2, \ldots$):**
$PA := \{v \mid v \in Proper$ and no $(u, *)$ is in *Locked* with $u \neq v\}$ ▷ *proper and acceptable*
broadcast $\langle$LIST, k, PA, *Proper*, *input*, *decision*$\rangle$
deliver set M of messages; call calc_proper(M) ▷ *same calc_proper as in Alg. 3*
add M to H; call check_dec(H)
▷ *check_dec called frequently on H due to possible lag in delivery of broadcast messages*

second superround of phase k**:**
if $p_i = leader(k)$ **then**
    $W := \{v \in V \mid$ there exist $n - t$ $\langle$LIST,$k, S, *, *, *\rangle$ messages in M with $v \in S\}$
    if $|W| > 0$ **then**
        pref := arbitrary element of W
        broadcast $\langle$LOCK, k, v, *Proper*, *input*, *decision*$\rangle$
deliver set M of messages; call calc_proper(M)
add M to H; check_dec(H)

third superround of phase k**:**
$S := \{v \in V \mid H$ contains a $\langle$LOCK$, v, k\rangle$ message from $leader(k)$ and $n - t$ $\langle$LIST$, k, T, *, *, *\rangle$ messages from different senders with $v \in T\}$
for each $v \in S$ **do**
    update (v, k) in *Locked*
if $|S| > 0$ **then**
    broadcast $\langle$ACK, k, S, *Proper*, *input*, *decision*$\rangle$ to $leader(k)$
    ▷ *other recipients are to ignore this message*
deliver set M of messages; call calc_proper(M) ▷ *only $leader(k)$ has $|M| > t$*
add M to H; call check_dec(H)
if (at least $2t+1$ ACK messages for the same value, v, are in M) and (*decision*= ⊥) **then**
    decision := v
delete from *Locked* every entry with phase smaller than the largest phase number of any entry in *Locked* ▷ *lock release without any communication*

The classic proofs of the $(t+1)$-round lower bound for consensus in the synchronous model with crash failures (e.g., [12, 20], see [8, Theorem 5.7]) construct a sequence of executions in which adjacent executions are *indistinguishable* to at least $n-1$ of the processes (meaning that each of the processes has the same history in the two executions), starting with an execution that decides 0 and ending with an execution that decides 1. Essentially the same construction can be applied to synchronized executions, resulting in the following:

▶ **Lemma 6.** *Consider any consensus algorithm that ensures weak unanimity for n processes that tolerates t Byzantine failures (possibly using authentication) where $n > t+1$. Let $s \leq t$. There exists a chain $\alpha_1, \alpha_2, \ldots, \alpha_m$ of s-round synchronized execution prefixes, for some m, such that:*

(a) *No process fails in α_1 (resp., α_m) and all the input values are 0 (resp., 1).*

(b) *For all i, $1 \leq i < m$, the number of processes that fail in either α_i or α_{i+1} (or both) is at most s.*

(c) *For all i, $1 \leq i < m$, there exists a process q such that α_i and α_{i+1} are indistinguishable to all processes except possibly q and one of the following holds:*

- **(1)** *the same processes are faulty in α_i and α_{i+1}, or*
- **(2)** *the only difference between the faulty processes in α_i and α_{i+1} is that q fails in α_{i+1} but not in α_i, or*
- **(3)** *the only difference between the faulty processes in α_i and α_{i+1} is that q fails in α_i but not in α_{i+1}.*

▶ **Lemma 7.** *Assume $n = 3t - 2s$, for some $s < t$. Any consensus algorithm for n processes that tolerates t Byzantine failures (possibly using authentication) and ensures weak unanimity has an execution in which every message has delay at most d but at least one correct process does not decide before time $s \cdot d + \Delta$.*

Proof. Let A be any consensus algorithm for n processes and t authenticated Byzantine failures. Let α be an s-round synchronized execution prefix of A. A *partition for* α is a partition (X, Y, F_1, F_2) of the processes such that $|X| = |Y| = |F_2| = t - s$, $|F_1| = s$, and F_1 contains all the processes that fail in α; since $t > s$, X and Y are nonempty. Such a partition is possible since $3(t-s) + s = 3t - 2s = n$. Note that $|F_1| + |F_2| = |F_1| + |X| = |F_1| + |Y| = t$.

The X-*extension* of α is the execution that extends α in which

- processes in $X \cup F_2$ remain correct,
- every process in $Y \cup F_1$ (that has not already failed) fails at the beginning of the extension (giving $t - s + s = t$ failures),
- failed processes behave by sending no messages, and
- every message sent after the end of α has delay d.

By the correctness of A, the correct processes must eventually decide in the extension; denote the decision value by $dec_X(\alpha)$.

Define the Y-*extension* of α and $dec_Y(\alpha)$ analogously, by swapping the roles of X and Y.

▷ Claim 8. For every s-round synchronized execution prefix α and every partition (X, Y, F_1, F_2) for α, $dec_X(\alpha) = dec_Y(\alpha)$.

Proof. Suppose in contradiction there exist α and (X, Y, F_1, F_2) such that $dec_X(\alpha) \neq dec_Y(\alpha)$. Let α_X (resp., α_Y) be the X-extension (resp., Y-extension) of α. By construction all message delays in α_X and α_Y are at most d. Let t_{dec} be the latest time at which some process in X decides in α_X or some process in Y decides in α_Y.

Assume in contradiction to the claim of the lemma that $t_{dec} < s \cdot d + \Delta$, so the decision is before Δ time has elapsed in the extension after the end of α. Now consider the extension β of α in which

- any process in F_1 that hasn't failed in α fails just after α, by sending no more messages;
- all processes in F_2 fail at the end of α by behaving in a "two-faced" manner until time $s \cdot d + \Delta$: toward X as they did in α_X and toward Y as they did in α_Y; after time $s \cdot d + \Delta$, processes in F_2 send no messages;
- all processes in $X \cup Y$ remain correct,
- every message sent after α within X or within Y has delay d, and
- every message sent after α between X and Y has delay Δ.

The next subclaim, which is the heart of the argument, shows that this behavior is allowed even if an authentication scheme is available.
Subclaim: For all t, $(s-1)d \leq t < (s-1)d + \Delta$,

1. Every process p_i in X (resp., Y) sends the same set of messages in step t of β as it does in step t of α_X (resp., α_Y).
2. Every process p_i in F_2 sends the same set of messages to processes in X (resp., Y) in step t of β as it does in step t of α_X (resp., α_Y).
3. Every process p_i in X (resp., Y) receives the same set of messages in step $t+1$ of β as it does in step $t+1$ of α_X (resp., α_Y).
4. Every process p_i in F_2 receives the same set of messages from processes in X (resp., Y) as it does in step $t+1$ of α_Y (resp., α_Y).
5. Every process p_i in X (resp., Y) has the same state in step $t+1$ of β as it does in step $t+1$ of α_X (resp., α_Y).

Proof of Subclaim: By induction on t.

Suppose $t = (s-1)d$. Consider $p_i \in X \cup Y \cup F_2$. The set of messages that p_i sends in step t of β is determined by p_i's state at the end of α, which is the immediately preceding prefix of β, α_X, and α_Y. Thus Properties (1) and (2) hold.

Property (3) holds since the set of messages S to a process in X (resp., Y) that are in transit just before step $t+1$ of β is a superset of that S' in α_X (resp., α_Y), thanks to Properties (1) and (2). The potential messages in S that are not in S' are those from Y to X (resp., X to Y), since all processes in Y (resp., X) are faulty in α_X (resp., α_Y) and send no more messages. However, the same set of messages are delivered because of how the message delays are specified and the fact that $t+1 \leq (s-1)d + \Delta$.

Property (4) holds since the same set of messages to a process in F_2 from a process in X (resp., Y) are in transit just before step $t+1$ of β as in α_X (resp., α_Y), thanks to Properties (1) and (2), and since the delays are specified to be the same.

Property (5) holds since a process in X (resp., Y) receives the same set of messages in step $t+1$ of β as in α_X (resp., α_Y) by Property (3) and its state in step t of β is the same as in step t of α_X (resp., α_Y) by the fact that both executions have α as a prefix, which ends with the state at time t.

Suppose $t \geq (s-1)d + 1$. Essentially the same argument as for $t = (s-1)d$ holds, with the following differences. Properties (1) and (5) rely on the inductive hypothesis for Property (5) (states being the same) instead of the fact that the executions have the same prefix. Finally, the crucial part of the proof is showing Property (2). This relies on the fact that, by the inductive hypothesis, each process in F_2 has received the same set of messages so far in β from processes in $X \cup Y$ as it does in α_X and in α_Y, so the authentication mechanism cannot prevent the faulty processes in β from sending the same set of messages to X (resp., Y) as in α_X (resp., α_Y).
End of proof of Subclaim.

The total number of failures in β is $|F_1| + |F_2| = s + (t - s) = t$. Up to time t_{dec}, processes in X cannot distinguish β and α_X and decide $dec_X(\alpha)$, while processes in Y cannot distinguish β and α_Y and decide $dec_Y(\alpha)$, contradicting agreement. ◁

Thanks to Claim 8, we can simply refer to the common value of $dec_X(\alpha)$ and $dec_Y(\alpha)$ as the "decision of the partition for an execution" when the partition is clear from the context.

Let $\alpha_1, \alpha_2, \ldots, \alpha_m$ be the chain of s-round synchronized execution prefixes from Lemma 6. Since all the inputs in α_1 are 0 and there are no failures, the weak unanimity condition implies that every partition for α_1 has decision 0. Similarly, every partition for α_m has decision 1. So there must be some i, $1 \leq i < m$, such that all partitions for α_i have decision 0 while at least one partition for α_{i+1} has decision 1, call it (X, Y, F_1, F_2).

Let q be a process such that α_i and α_{i+1} are indistinguishable to all processes except possibly q (q exists by Lemma 6). We consider the three cases of how α_i, α_{i+1} and q are related from Lemma 6, and show a contradiction for each one, implying that t_{dec} must be at least $s \cdot d + \Delta$.

Case 1: The same processes are faulty in α_i and in α_{i+1}. If q is not in X, then α_i and α_{i+1} are indistinguishable to all processes in X. Since the partition (X, Y, F_1, F_2) has decision 1 for α_{i+1}, it also has decision 1 for α_i, which is a contradiction. If q is in X, then it is not in Y and the analogous argument holds.

Case 2: The only difference between the faulty processes in α_i and α_{i+1} is that q fails in α_{i+1} but not in α_i. Then q is in F_1 so α_i and α_{i+1} are indistinguishable to all processes in $X \cup Y$. So (X, Y, F_1, F_2) has decision 1 for α_i, a contradiction.

Case 3: The only difference between the faulty processes in α_i and α_{i+1} is that q fails in α_i but not in α_{i+1}.

If q is in $F_1 \cup F_2$, then the same argument as in Case 2 holds.

Suppose q is not in $F_1 \cup F_2$; without loss of generality, q is in Y. There must be some process p such that p is in F_1 but not faulty in α_i. The reason is that $|F_1| = s$, at most s processes are faulty in α_i (by part (b) of Lemma 6), and there is a faulty process in α_i not in F_1 (namely, q).

Thus q is in Y but not in F_1, p is in F_1 but not in Y, p is not faulty in α_i, and α_i and α_{i+1} are indistinguishable to all processes in X. Thus $(X, (Y - \{q\}) \cup \{p\}, (F_1 - \{p\}) \cup \{q\}, F_2)$ is a partition for α_i with decision 1, which is a contradiction. ◀

Note that Lemma 7 applies for any $n \leq 3t$. When $n = 3t$, then $s = 0$ and the bound is Δ; when $n = 3t - 2$, then $s = 1$ and the bound is $d + \Delta$; when $n = 2t$ (and t is even), then $s = t/2$ and the bound is $(t/2)d + \Delta$; finally, when $n = t + 2$, then $s = t - 1$ and the bound is $(t - 1)d + \Delta$. In general, we have:

▶ **Theorem 9.** *Assume $n \leq 3t$. Any consensus algorithm for n processes that tolerates t Byzantine failures (possibly using authentication) and ensures weak unanimity has an execution in which every message has delay at most d but at least one correct process does not decide before time $\lfloor (3t - n)/2 \rfloor d + \Delta$.*

6 Discussion

This paper studies whether the cost of timeout can be avoided when solving agreement problems in the bounded-delay model. On the positive side, we prove that the consensus problem can be solved with asymptotically optimal time of $O(t \cdot d)$, in the presence of t failures, for send omission, authenticated and unauthenticated Byzantine failures. For send omission failures, our algorithm requires n to be greater than $2t$, while the Byzantine algorithms

require n to be greater than $3t$. On the negative side, we show that dependence on Δ cannot be avoided if $n \leq 3t$ for authenticated Byzantine failures. Specifically, when $n \leq 3t$, the time complexity for consensus with weak unanimity is at least $\lfloor 3t - n)/2 \rfloor \cdot d + \Delta$. The results show that the cost of timeouts can be avoided for these agreement problems if and only if the resilience is the same as that needed for solving consensus in eventually-synchronous systems.

An immediate open question is to find the optimal time complexity when $n \leq 3t$ for authenticated Byzantine failures. In this direction, note that if $3t - n$ faulty processes are taken out, then the remaining faulty processes constitute less than a third of the remaining system. This might indicate a path to finding an upper bound that matches our lower bound. There are analogous open questions for the case of crash and (send) omission failures when $n \leq 2t$. Interestingly, the crash lower bound when $n \leq 2t$ has a $2t - n$ term, which corresponds to the number of failed processes that should be taken out in order to have a majority of correct processes.

A challenging research direction is to study the time complexity of consensus algorithms in the *eventually-synchronous* model [13], where the upper bound of Δ on message delay only holds after GST. Numerous consensus algorithms have been proposed for this model, e.g., [10, 30], but to the best of our knowledge, their time complexity is in $\Omega(t \cdot \Delta)$ *after GST*.

References

1 Ittai Abraham. State machine replication for two servers and one omission failure is impossible even in a lock-step model. https://decentralizedthoughts.github.io/2019-11-02-primary-backup-for-2-servers-and-omission-failures-is-impossible/, 2019.

2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020. doi:10.1109/SP40000.2020.00044.

3 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Brief announcement: Byzantine agreement, broadcast and state machine replication with optimal good-case latency. In *34th International Symposium on Distributed Computing (DISC)*, 2020. arXiv:2003.13155.

4 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of Byzantine broadcast: A complete categorization. In *2021 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 331–341, 2021. doi:10.1145/3465084.3467899.

5 Hagit Attiya, Fatemeh Borran, Martin Hutle, Zarko Milosevic, and André Schiper. Structured derivation of semi-synchronous algorithms. In *International Symposium on Distributed Computing (DISC)*, pages 374–388. Springer, 2011. doi:10.1007/978-3-642-24100-0_37.

6 Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *J. ACM*, 41(1):122–152, 1994. doi:10.1145/174644.174649.

7 Hagit Attiya, Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, pages 359–369. ACM, 1991. doi:10.1145/103418.103457.

8 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. McGraw-Hill Publishing Company, 1st edition, 1998.

9 A.A. Bharali and P. Berman. Distributed consensus with general omission failures and timing uncertainty. In *Proceedings of Phoenix Conference on Computers and Communications*, pages 168–174, 1993. doi:10.1109/PCCC.1993.344468.

10 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002. doi:10.1145/571637.571640.

11 Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine consensus is $\Theta(n^2)$: The Dolev-Reischuk bound is tight even in partial synchrony! In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 14:1–14:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.14.

12 Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. doi:10.1137/0212045.

13 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.

14 Cynthia Dwork and Larry Stockmeyer. Bounds on the time to reach agreement as a function of message delay. Technical Report RJ 8181 (75140), IBM Thomas J. Watson Research Division, jun 1991. available in https://drive.google.com/file/d/1zoCe05kzhOWBRi2DQ7OmFpZJxBPMpzSS/view?usp=sharing.

15 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986. doi:10.1007/BF01843568.

16 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.

17 Juan A Garay, Jonathan Katz, Chiu-Yuen Koo, and Rafail Ostrovsky. Round complexity of authenticated broadcast with a dishonest majority. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 658–668, 2007. doi:10.1109/FOCS.2007.61.

18 Amir Herzberg and Shay Kutten. Fast isolation of arbitrary forwarding faults. In Piotr Rudnicki, editor, *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989*, pages 339–353. ACM, 1989. doi:10.1145/72981.73006.

19 Amir Herzberg and Shay Kutten. Early detection of message forwarding faults. *SIAM Journal on Computing*, 30(4):1169–1196, 2000. doi:10.1137/S0097539796312745.

20 Michael Merritt. Notes on the Dolev-Strong lower bound for Byzantine Agreement. unpublished manuscript, 1985.

21 Dimitris Michailidis. Fast set agreement in the presence of timing uncertainty. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, PODC, '99Atlanta, Georgia, USA, May 3-6, 1999*, pages 249–256. ACM, 1999. doi:10.1145/301308.301365.

22 Atsuki Momose, Jason Paul Cruz, and Yuichi Kaji. Hybrid-BFT: Optimistically responsive synchronous consensus with optimal latency or resilience. Cryptology ePrint Archive, Paper 2020/406, 2020. URL: https://eprint.iacr.org/2020/406.

23 Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *31 International Symposium on Distributed Computing (DISC)*, 2017. doi:10.4230/LIPICS.DISC.2017.39.

24 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology (EUROCRYPT): 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2018. doi:10.1007/978-3-319-78375-8_1.

25 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

26 Stephen Ponzio. The real-time cost of timing uncertainty: Consensus and failure detection. Master's thesis, MIT, 1991.

27 Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 839–857, 2020. doi:10.1145/3372297.3417284.

28 TK Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987. doi:10.1007/BF01667080.

29 Jun Wan, Hanshen Xiao, Elaine Shi, and Srinivas Devadas. Expected constant round Byzantine broadcast under dishonest majority. In *18th International Conference on Theory of Cryptography (TCC)*, pages 381–411, 2020. doi:10.1007/978-3-030-64375-1_14.

30 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.

Black Hole Search in Dynamic Rings: The Scattered Case

Giuseppe A. Di Luna
DIAG, Sapienza University of Rome, Italy

Paola Flocchini
School of Electrical Engineering and Computer Science, University of Ottawa, Canada

Giuseppe Prencipe
Department of Computer Science, University of Pisa, Italy

Nicola Santoro
School of Computer Science, Carleton University, Ottawa, Canada

Abstract

In this paper we investigate the problem of searching for a black hole in a dynamic graph by a set of scattered agents (i.e., the agents start from arbitrary locations of the graph). The black hole is a node that silently destroys any agent visiting it. This kind of malicious node nicely models network failures such as a crashed host or a virus that erases the visiting agents. The black hole search problem is solved when at least one agent survives, and it has the entire map of the graph with the location of the black hole. We consider the case in which the underlining graph is a dynamic 1-interval connected ring: a ring graph in which at each round at most one edge can be missing. We first show that the problem cannot be solved if the agents can only communicate by using a face-to-face mechanism: this holds for any set of agents of constant size, with respect to the size n of the ring.

To circumvent this impossibility we consider agents equipped with movable pebbles that can be left on nodes as a form of communication with other agents. When pebbles are available, three agents can localize the black hole in $O(n^2)$ moves. We show that such a number of agents is optimal. We also show that the complexity is tight, that is $\Omega(n^2)$ moves are required for any algorithm solving the problem with three agents, even with stronger communication mechanisms (e.g., a whiteboard on each node on which agents can write messages of unlimited size). To the best of our knowledge this is the first paper examining the problem of searching a black hole in a dynamic environment with scattered agents.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Theory of computation → Self-organization

Keywords and phrases Black hole search, mobile agents, dynamic graph

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.33

Funding The work was partially supported by the Natural Sciences and Engineering Research Council of Canada under Discovery Grants and by "PRA – Progetti di Ricerca di Ateneo" (Institutional Research Grants) – Project no. PRA_2022_81 of the Pisa University and PRA RM1221816C1760BF from Sapienza and by (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union.

1 Introduction

1.1 Exploration of Dynamic Networks

In the distributed computing community a large set of works (see [20]) studies the computational paradigm of *mobile agents*. A *mobile agent* is a software component that is able to move on a network visiting nodes. When a node is visited, the agent executes some

computation interacting with the local environment, the memory and resources of the visited computational node, and then it moves on a next computational node by transmitting itself on the network. That is the agent can be seen as an intelligent message with computational capabilities and that is able to decide its next destination.

In the mobile agent paradigm a plethora of problems have been studied. The most famous are: *exploration*, the agents have to collectively visit the entire network; *gathering*, the agents have to reach the same node; *patrolling*, the agents have to periodically patrol the network minimising the time between two visits of the same node.

One problem that has been thoroughly investigated is the Black Hole Search (BHS) [25]. In this problem there exists a dangerous stationary node, called *black hole* (BH), that silently erases from the network all the agents that visit it. A BH node could model several kind of common failures: for instance, consider a crashed host, and any agent trying to visit it will be lost and removed from the network; or a node infected by a virus that cancels the incoming agents. The many works that investigated BH have given to us a complete knowledge of the computational properties of the problem under several assumptions (examples are communication mechanisms employed by the agents, level of synchronicity, topology of the network, and agents' knowledge and capabilities). However, almost all of them examine the case in which the network is *static*: the set of computational nodes and the links connecting them are static and never change.

Recently, research within distributed computing has started to focus on mobile-agents in *highly dynamic graphs*, i.e., graphs where the topological changes are not limited to sporadic and disruptive events (such as process failures, links congestion, etc). Highly dynamic graphs model a wide range of modern networked systems whose dynamic nature is the natural product of innovations in communication technology (e.g., wireless networks), in software layer (e.g., a controller in a software defined network), and in society (e.g., the pervasive nature of smart mobile devices) (e.g., see [5]).

We consider *evolving graphs*, that are dynamic graphs that can be seen as an infinite sequence of static graphs. In this case the model of computation is by definition *synchronous*, and at each round corresponds a static graph of the aforementioned sequence. A popular assumption in this model is the *1-interval connectivity*: this assumption dictates that at each round the dynamic graph is connected (e.g., [1, 10, 21–23, 26]).

We focus on the case of 1-interval connected rings, where the topology is a ring graph in which at each round at most one edge is missing.

In the last years the body of works studying agents on highly dynamic graphs has been growing with a sustained velocity (for a recent survey see [9]). In particular, a large number of papers is focussing on *1-interval connected rings*: the *gathering* problem has been investigated in [12], the *exploration* problem in [3,4,11] and the BHS for colocated[1] agents in the recent [13]. Despite this large interest, a lot of questions are open. One of them is answered in this paper: how does the computational landscape of finding a BH change when agents are *scattered*? In the scattered case agents start from arbitrary nodes of the graph. We will show that this setting has many differences with respect to the case of colocated agents (studied in [13]) in terms of both solvability (solving BHS in some settings becomes impossible) and complexity.

[1] Agents are colocated when they all start from the same node.

1.2 Related Works

The Black Hole Search (BHS) problem has ben introduced in [15]. The problem has been studied in graph of restricted topologies (e.g., trees [8], rings and tori [6, 17, 24]) and in arbitrary and possibly unknown topology (e.g., [7, 14, 15]). For a recent survey see [25]. The most relevant papers are the ones investigating BHS in static ring networks. In the *asynchronous* setting, it is possible to solve the problem with two colocated agents and $\Theta(n \log n)$ moves, in the whiteboard model [16], and in the pebble model [18]. It has been shown that $\mathcal{O}(n \log n)$ moves also suffices for the scattered case and oriented rings [6]. Others [2] investigated time-optimal algorithms when considering unitary delay.

In spite of all the differences in settings and assumptions, all these investigations share a common trait: the agents operate on a *static* network.

The only works studying BHS in dynamic graph are [13] and [19]. [19] is on the black hole search in *carrier graphs*, a particular class of periodic temporal graphs defined by circular intersecting routes of public carriers, where the stops are the nodes of the graph and the agents can board and disembark from a carrier at any stop.

[13] studies the BHS in the same setting studied in this paper: 1-interval connected ring with a single BH. [13] shows that three agents are necessary to find the BH (in the static case two agents are enough to explore arbitrary known graphs [7]), and it presents optimal algorithms to find the BH with three agents. Moreover, if agents can communicate only when they are on the same node, the authors show that BHS can be solved in $\Theta(n^2)$ moves and rounds. Finally, if agents can use pebbles, they show an improved algorithm that finds the BH in $\Theta(n^{1.5})$ moves and rounds.

In both studies, the agents are assumed to be initially *colocated*, i.e. to start from the same safe node. To the best of our knowledge, no study considers the case of scattered agents.

1.3 Contributions

We study the problem of finding a BH in an oriented dynamic ring by a set of scattered agents with visible identities under different communication capabilities. We study two main families of communication mechanisms. The *endogenous* family, where the agents can communicate without using any external *facilities*. In this case, they can either see each other only when on the same node (*Vision model*), or they can also talk with each other (*Face2Face model*). In contrast, in the *exogenous* family, the agents communicate using external tools. In this case, we have the *Pebbles model*, in which each agent carries a pebble that can leave on a node to mark it for other agents, or that it can remove from a node in case this marking is not needed anymore; or the *Whiteboard model*, in which each node has a public whiteboard on which agents can write messages of unlimited size.

Our first result (Obs. 6) is that in the endogenous family the BHS is unsolvable using three agents. This is in contrast with the colocated case where BHS is solvable in the same setting by using three agents (see [13]).

To circumvent such impossibility we then rely on *exogenous* communication. We focus on algorithms that use an optimal number of agents. In particular, in [13], it has been shown that BHS in dynamic rings is unsolvable if only two agents are available. This results clearly extends also to the scattered case. Therefore, we will consider algorithms for set of three agents (size optimal algorithms). In Th. 7, we show that any optimal size algorithm solving BHS requires $\Omega(n^2)$ moves and $\Omega(n^2)$ rounds in the whiteboard model. We note that on a static and synchronous rings two agents can find the blackhole in $O(n)$ moves and rounds.

This observation is interesting as it shows that dynamicity does not only increase the number of required agents but it also increases, significatively, the time and moves required. This also highlights the price to pay for having scattered agents: in fact, for dynamic rings, $O(n^{1.5})$ moves and rounds are sufficient in the colocated case [13]. Finally, our lower bound is tight: we provide an algorithm that solves BHS in the pebble model in $O(n^2)$ rounds and moves using three agents (Th. 17).

2 Model and Preliminaries

2.1 The Model and the Problem

The system is a temporal graph where a set of nodes V is connected by a set of edges E. The system is synchronous, and the dynamic networks is an evolving graph $\mathcal{G}$. The time is divided in fictional unites called rounds. The evolving graph can be seen as a sequence of static graphs: $\mathcal{G} = G_0, G_1, \ldots, G_r, \ldots$, where $G_r = (V_r, E_r)$ is the graph of the edges present at round r. The footprint of the dynamic graph is a static graph containing all the edges that will be present in the system, alternatively, is the union of all the graphs in the aforementioned sequence. An evolving graph where connectivity is guaranteed at every round is called *1-interval connected* (i.e., $\forall G_i \in \mathcal{G}$, G_i is connected). In this paper we focus on *dynamic rings*: 1-interval connected graphs whose footprint is a ring. Let $\mathcal{R} = (v_0, v_1, \ldots v_{n-1})$ be a dynamic oriented ring, i.e., where each node v_i has two ports, consistently labelled left and right connecting it to v_{i-1} and v_{i+1} (operations on indices are modulo n). Nodes are anonymous, that is they do not have IDs. A set $A = \{a_0, a_1, \ldots, a_{k-1}\}$ of mobile agents inhabits $\mathcal{R}$. The agents start from distinct arbitrary locations: they are *scattered*. Agents have distinct visible identifiers in $\{0, \ldots, k-1\}$ and they know the total size of the ring n. The agents can move from node to neighbouring node and they have bounded storage ($\mathcal{O}(\log n)$ bits of internal memory suffice for our algorithms). In each round all agents are activated. Upon activation, an agent on node v at round r takes a local snapshot of v that contains the set $E_r(v)$ of edges incident on v at this round, and the set of agents present in v. The agent communicates with the others (the communication mechanism employed will be discussed later). On the basis of the snapshot, the communication, and the content of its local memory, an agent then decides what action to take. The action consists of a *communication step* (defined below) and a *move step*. In the move step the agent may decide to stay still or to move on an edge $e = (v, v') \in E_r(v)$. In the latter case, if the edge is present, the agent will reach v' in round $r + 1$.

We consider two classes of communication mechanisms (*endogenous* and *exogenous*) which give rise to four models.

Endogenous Mechanisms rely only on the robots' capabilities without requiring any external object. In the *FaceToFace (F2F)* model the agent can explicitly communicate among themselves only when they reside on the same node. In the *Vision* model an agents can see all the other agents that reside on the same node (hence count their number); however, they cannot communicate.

Exogenous Mechanisms do require external objects for the robots to exchange information. Among those we distinguish:

- *Pebble:* each agent is endowed with a single pebble that can be placed on or taken from a node. On each node, the concurrent actions of placing or taking pebbles are done in fair mutual exclusion.
- *Whiteboard:* each node contains a local shared memory, called whiteboard, of size $O(\log n)$ where agents can write on and read from. Access to the whiteboard is done in adversarial but fair mutual exclusion.

The temporal graph $\mathcal{G}$ contains a *black hole* (BH), a node that destroys any visiting agent without leaving any detectable trace of that destruction.

We say that an agent knows the footprint of $\mathcal{R}$ when it knows its left and right distance from the blackhole, that is the agent is able to build in its memory a graph isomorphic to the footprint of $\mathcal{R}$ and it knows its relative position with respect to the blackhole.

▶ **Definition 1** (BHS [13]). *Given a dynamic ring $\mathcal{R}$, and an algorithm $\mathcal{A}$ for a set of agents we say that $\mathcal{A}$ solves the* BHS *if at least one agent survives and terminates knowing the footprint of $\mathcal{R}$. Each agent that terminates has to know the footprint of $\mathcal{R}$.*

Since n is known it is enough that the agent knows its right (or left) distance from the blackhole in order to know the footprint, this means that is not necessary for the agent to visit all nodes of $\mathcal{R}$.

We call *size* the number of agents used by the protocol. Other measures of complexity are the total number of moves performed by the agents, which we shall call *cost*, and *time* it takes to complete the task.

Figure 1 shows (a) four rounds of an execution in a dangerous dynamic ring, and (b) the space diagram representation that we will use in this paper. The agent is represented as the black quadrilateral and it is moving clockwise; the BH is the black node. At round $r = 2$ and $r = 3$ the agent is blocked by the missing edge. In the diagram, the movement of the agent is represented as a solid line.

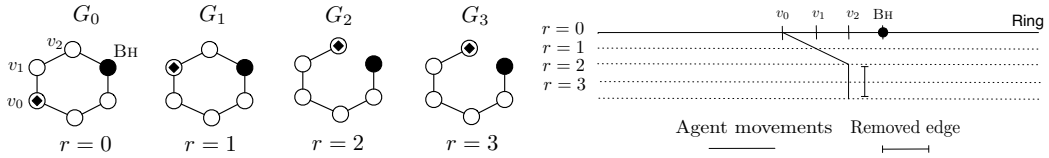


Figure 1 (a) Execution in a dangerous dynamic ring, and (b) its space diagram representation.

3 Preliminaries

Before presenting and analyzing our solution protocols, we report some known impossibility results and a technical lemma that we will use in our paper. Then we briefly describe a well known idea employed for BHS in static graphs that will be adapted in our algorithms, as well as the conventions and symbols used in our protocols.

3.1 Known Impossibilities

In this section we report known impossibility results.

▶ **Theorem 2** ([13]). *In a dynamic ring of size $n > 3$, two colocated agents cannot solve the* BHS*. The impossibility holds even if the agents have unique IDs, and are equipped with the strongest (*Whiteboard*) communication model.*

▶ **Theorem 3** ([13]). *There exists no algorithm that solves the* BHS *in a dynamic ring $\mathcal{R}$ whose size is unknown to the agents. The result holds even if the nodes have whiteboards, the agents have IDs, and irrespectively of the number of agents.*

▶ **Observation 4** ([13]). *Given a dynamic ring $\mathcal{R}$, and a cut U (with $|U| > 1$) of its footprint connected by edges e_c and e_{cc} to nodes in $V \setminus U$, agents in U explore a node outside the cut at the end of round r if and only if, in round r, there are two agents in U, one that tries to traverse e_c and one trying to traverse e_{cc}, respectively.*

3.2 Cautious Walk

Cautious Walk is a mechanism introduced in [15] for agents to move on dangerous graphs in such a way that two (or more) agents never enter the black hole from the same edge. The general idea of cautious walk in static graphs is that when an agent a moves from u to v through an unexplored (thus dangerous) edge (u, v), a must leave the information that the edge is under exploration at u. The information can be provided through some form of mark in case of exogenous communication mechanisms, or implicit in case of endogenous mechanisms (e.g., by employing a second agent as a "witness"). In our algorithms we will make use of variants of the general idea of cautious walk, adapting it to the dynamic scenario.

3.3 Pseudocode Convention and Communication

We use the pseudocode convention introduced in [11]. In particular, our algorithms use as a building block procedure Explore ($dir \mid p_1 : s_1; p_2 : s_2; \ldots; p_k : s_k$), where $dir \in \{left, right, nil\}$, p_i is a predicate, and s_i is a state. In Procedure Explore, the agent takes a snapshot, then evaluates predicates $p_1, \ldots, p_k$ in order; as soon as a predicate is satisfied, say p_i, the procedure exits, and the agent transitions to the state s_i specified by p_i. If no predicate is satisfied, the agent tries to move in the specified direction dir (or it stays still if $dir = nil$), and the procedure is executed again in the next round. The following are the main predicates used in our Algorithms:

- *meeting*[ID/Role]: the agent sees another agent with identifier ID (or role *Role*) arriving at the node where it resides, or the agent arrives in a node, and it sees another agent with identifier ID (or role *Role*).

Furthermore, the following variables are maintained by the algorithms:

- $Ttime$, $Tnodes$: the total number of rounds and distinct visited nodes[2], respectively, since the beginning of the execution.
- $Etime$, $Enodes$: the total number of rounds and distinct visited nodes, respectively, since the last call of procedure Explore.
- $EMtime$ [C/ (CC)]: the number of rounds during which the clockwise/ (resp. counter-clockwise) edge is missing since the last call of procedure Explore.
- $\#Meets[ID]$: the number of times the agent has met with agent ID.
- $RLastMet[ID]$ records the number of rounds elapsed since the agent has seen (or meet) an agent with id ID

Observe that, in a fully synchronous system, when predicate *meeting*[y] holds for an agent a with id x, then predicate *meeting*[x] holds for the agent with id y. In the pebble model we will also use the CautiousExplore procedure: it is analogous to Explore, with the main difference that the agent uses the pebble to perform a cautious walk. That is, the agent leaves a pebble on the current node, it moves to the node in the moving direction, then it goes back to remove the left pebble, and finally it returns to the recently explored node.

[2] An agent is able to identify if the node where it resides has been previously visited or not by counting the number of steps it has performed in certain direction.

Communication and pebble removal. As for the ability of agents to interact, we observe that, even in the simpler pebble model, any communication between agents located at the same node is easy to achieve (e.g., two agents may exchange messages of any size using a communication protocol in which they send one bit every constant number of rounds). Therefore, we can assume that, in the exogenous models, agents are able to communicate. Specifically, the communication of constant size messages is assumed to be instantaneous, since it can implemented trivially by a multiplexing mechanism (the logical rounds are divided in a constant number of physical round, the first of which is used to execute the actual algorithm and the others to communicate). During a cautious walk using CautiousExplore procedure, agent x might return to a node to retrieve its pebble, even while another agent is present on the same node. In such instances, the two agents meet, but any state transitions initiated by this meeting will only take effect after agent x has successfully reclaimed its pebble.

3.4 CautiousPendulum: An algorithm for colocated agents

In this section we describe the Bhs algorithm CautiousPendulum presented in [13]. The algorithm solves the Bhs when three agents with visible IDs start from the same node. We will use CautiousPendulum as subprocedure in our algorithm for the scattered case (Section 6).

The CautiousPendulum algorithm uses three agents: the Avanguard, the Retroguard, and the Leader. The three agents start on the home-base node v_0.

Agents Avanguard and Leader move clockwise simulating a cautious walk. In particular, If the edge e in the clockwise direction is not present, both agents wait until it reappears. If edge e is present, Avanguard moves to the unexplored node using edge e. Then, if in a successive round the edge e is still present, Avanguard goes back to Leader, signalling that the visited node is safe; at this point, both Leader and Avanguard move clockwise to the recently explored node. If Avanguard does not return when e is present, then the Leader knows that the node visited by Avanguard is the blackhole.

Retroguard moves as follows: it goes counter-clockwise until it visits the first unexplored node; then, it goes back clockwise until it meets again Leader. Once Retroguard meets Leader, it moves again counter-clockwise, iterating the same pattern. In case Retroguard finds a missing edge on its path, it waits until the edge re-appears; then it resumes its movement.

If the Leader sees a missing edge e in its clockwise direction and, while waiting for e to appear, does not meet Retroguard after enough time for Retroguard to explore a node and go back, then we say that Agent Retroguard *fails to report* to Leader. In this case, Retroguard entered the black hole, hence the Leader can correctly compute its location.

▶ **Theorem 5** ([13]). *Consider a dynamic ring $\mathcal{R}$, with three colocated agents with distinct IDs in the Vision model. Algorithm* CautiousPendulum *solves* Bhs *with $\mathcal{O}(n^2)$ moves and in $\mathcal{O}(n^2)$ rounds.*

4 Impossibility with Scattered Agents and Endogenous Communication

When the agents are scattered, three of them, even equipped with the stronger endogenous mechanism (i.e., F2F model), cannot solve Bhs on rings of arbitrary size, as shown by the following:

▶ **Observation 6.** *Three scattered agents in the F2F model cannot solve* BHS *on a static ring of arbitrary size n, even if they have distinct IDs.*

Proof. Let $\mathcal{A}$ be an algorithm that correctly solves the problem. The proof is by contradiction: we will show that there exists an initial configuration C of the 3 agents on a ring of a proper size $n > 10$ that makes $\mathcal{A}$ fail. We will construct the configuration C in such a way that no two agents meet. Let id_1, id_2, id_3 be the IDs of agents. We consider the behaviour of agent id_i until round r in a run where it executes the algorithm $\mathcal{A}$, and it does not meet any agent. Let $D(a_i, n, r)$ the maximum distance in any direction travelled by an agent until round r. Let r_m be the minimum round at which $D(a_i, n, r) > 0$ for some a_i. Without loss of generality, let id_1 be this agent. We can position the agents so that id_1 is adjacent to the blackhole and enters it at round r_m. Every other agent can be positioned such that they have not met any other agent by round r_m. At this stage, we are left with two agents, and we can invoke Th. 2 to conclude that it is impossible to solve the problem. Note that the premise of Th. 2 is based on co-located agents. However, this scenario is simpler than the dispersed case, so the result is applicable to our context as well. ◀

Fortunately, any exogenous mechanism circumvents the impossibility of Obs. 6. In the following we focus on such mechanisms.

5 Exogenous Communications: Lower Bound for Size-optimal Algorithms

We now consider the Exogenous Communications. Interestingly, we can show a quadratic lower bound on the number of moves and rounds of any size-optimal algorithm that solve that BHS-Problem with scattered agents; the bound holds even if agents have IDs and whiteboards are present.

▶ **Theorem 7.** *In a dynamic ring $\mathcal{R}$ with whiteboards, any algorithm $\mathcal{A}$ for* BHS *with three scattered agents with unique IDs requires $\Omega(n^2)$ moves and $\Omega(n^2)$ rounds.*

Proof. The proof is by contradiction. Let $\mathcal{A}$ be a sub-quadratic algorithm that solves the BHS-Problem; and let a, b, and c be the three agents. Suppose the agents have unique IDs, and, without loss of generality, let c be the first agent that moves.

Let us assume an initial configuration where c is initially placed on node v_c, neighbour of the Bh, and where a and b are on two neighbours nodes, v_a and v_b, at distance $n/2$ from v_c. Furthermore, w.l.o.g, let us assume that c is placed in such a way that when it moves, at round $r = 0$, it immediately enters Bh; also, let v_c be the counter-clockwise neighbor of the Bh.

Note that, (N1) since c enters the blackhole at round 0 and it can write information on node v_c (whiteboard model), agents a and b can compute the position of the Bh only in two cases: either (N1.1) one of them enters Bh or (N1.2) one of them visits node v_c. We now prove that case (N1.2) might never happen. Let U_r be the partition of nodes explored by agents a and b at the end of round r. By Observation 4, agents a and b may explore a node outside U_r only if they try to traverse at the same round both the edges crossing the cut U_r and $V \setminus U_r$. Let e_c be the clockwise edge incident in U_r, and e_{cc} be the counter-clockwise one. Edge e_c might always be missing, thus preventing the agents from crossing it. Therefore, Bh might only be reached by its clockwise neighbor, and node v_c will never be explored.

Since the blackhole is at distance $\frac{n}{2}$ from both v_a and v_b, by (N1.1) there must exists a set of rounds $r_1, r_2, \ldots, r_{n/8}$ where $U_{r_j} \geq n/8 + j$. We now prove that if $\mathcal{A}$ is sub-quadratic, then in one of these rounds, there must exist an agent, say b, that explores at least two

nodes, say v_1 at round r_1 and v_2 at round r_2, such that (1) it does not communicate with a between the two explorations and (2) both a and b visit $o(n)$ disjoint nodes between r_1 and r_2. Assume by contradiction then neither (1) or (2) apply. Then, in each U_{r_j} agent b explores only one node and the agents collectively performs at least $n/8$ moves. Since there are $n/8$ such rounds we have at least $\mathcal{O}(n^2)$ moves. Thus having a contradiction.

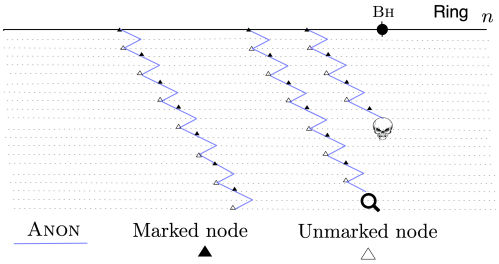
Note that, since the initial configuration is arbitrary and a and b never received any information communicated from c, the positions of v_1 and v_2 do not depend on the positions of BH and v_c. Therefore, there can exist two initial configurations C_1 and C_2, such that $v_1 = Bh$ in C_1 and $v_2 = Bh$ in C_2. Since b reaches the BH by round r_2, a is the only agent that can disambiguate between the two configurations. However, a might be blocked indefinitely on a set of nodes that was never visited by b after round r_1 (see Observation 4 – at round r_2 agent a is trying to traverse edge e_c). Consequently, a is not able to distinguish between C_1 and C_2; thus, $\mathcal{A}$ cannot be correct, having a contradiction. Finally, the bound on the number of rounds derives immediately from the bound on moves and from having a constant number of agents. ◀

The above theorem shows the cost optimality of the size-optimal algorithm Gather&Locate described in the following Section 6.

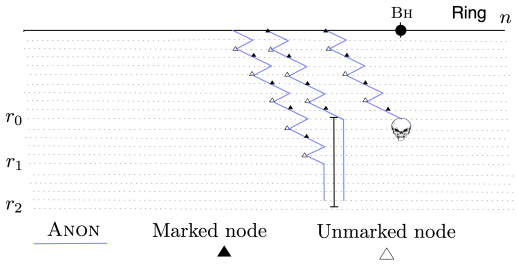
6 An Optimal Exogenous Algorithm: Gather&Locate

In this section, we describe an algorithm to solve the problem with $k = 3$ agents using pebbles. We name the algorithm Gather&Locate. Gather&Locate works in two phases:

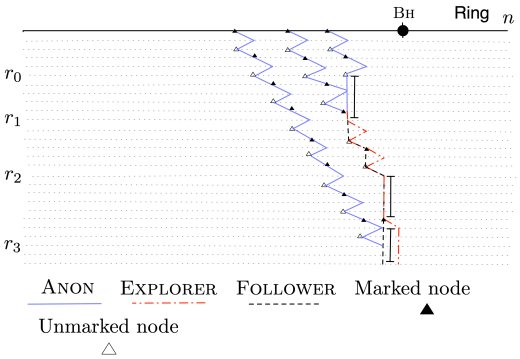
- **Phase 1:** In the first phase agents move clockwise using pebbles to implement a cautious walk. If they meet they synchronise their movements such that at most one of them enters in the black hole. This phase lasts until all three agents meet or $9n$ rounds have passed. We will show that at the end of this phase we have either:
  - (1) three agents are on the same node or on the two endpoint nodes of the same edge. In this case we say that agents gathered;
  - (2) the counter-clockwise neighbour of the black hole has been marked, at most one agent is lost, and the two remaining agents are gathered (that is they are on the same node, or on two endpoints of an edge);
  - (3) one agent is lost in the black hole and the counter-clockwise neighbour of the black hole has been marked. The remaining two agents either both terminated, locating BH, or only one terminated, with the other still looking for the BH. In Phase 2 this last agent will either terminate or it will be blocked forever (in both cases the problem is solved).
- **Phase 2:** The second phase starts after the previous one, and relies on the properties enforced by the first phase. In particular, if at the beginning of this phase three agents are on the same node, they start algorithm CautiousPendulum. Otherwise, if two agents are on the same node, they act similarly to RETROGUARD and LEADER in CautiousPendulum. If none of the above applies, then two agents are on the two endpoints of a missing edge, or only a single agent is still looking for the BH. This scenario is detected by a timeout strategy: upon a timeout, an agent starts moving clockwise looking for the node marked during Phase 1. If two agents meet during this process, they act similarly to RETROGUARD and LEADER in CautiousPendulum. Otherwise, in case a single agent is still active, it will either reach the marked node (and terminate correctly) or it will be blocked forever on a missing edge. We remark that in this last case there has been an agent correctly terminating in Phase 1, and thus BHS-PROBLEM is still correctly solved.



(a) Phase 1: run where BHS-PROBLEM is solved.



(b) Phase 1: run where two agents gather. At round r_0 the rightmost agent enters in the black hole, while the middle agent is blocked. At round r_1 the two remaining agents gathers.



(c) Phase 1: run where three agents gather. At round r_0 the rightmost agent is blocked. At round r_1 two agents meet creating a pair EXPLORER-FOLLOWER. At round r_2 the pair is blocked and the leftmost agent is able to catch up. At round r_3 the tree agents gathered. Note that the meeting predicate of the START with the FOLLOWER triggers at round r_3 and $r_3 - 2$: when an agent is cautious exploring it cannot meet other agents if it has to unmark a node.

Figure 2 Example of runs for Phase 1 of Gather&Locate.

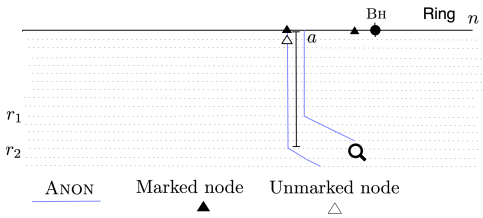
6.1 Detailed Description

The pseudocode of Phase 1 is reported in Algorithms 1, 3, and 2; and Phase 2 in Algorithms 4 and 5. In the pseudocode, we use the predicate $\#A = x$ that is verified when on the current node there are x agents. Initially, all agents have role START.

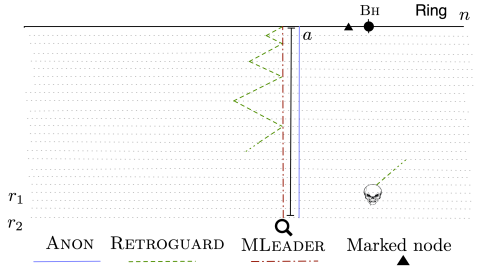
Phase 1. The first phase lasts for at most $9n$ rounds (refer also to the examples in Figure 2). The agents start in state Init of Algorithm 1 and role START: each agent walks cautiously clockwise for $9n$ rounds. If an agent reaches a marked node (predicate *marked*), then it waits until the next node can be deemed as safe or unsafe (see state Wait).

If in the marked node the incident clockwise edge is present and the agent that marked the node does not return, then the next node is the black hole (the agent terminates by triggering predicate $NextUnsafe$).

If two START agents meet on the same node (predicate *meeting*[START]), they synchronise their movements such that they will never cross an edge leading to a possibly unsafe node in the same round. Specifically, the agents enter in the synchronisation state Two, where one agent becomes FOLLOWER (Algorithm 3) and the other becomes EXPLORER (Algorithm 2).



(a) Phase 2: case of two agents gathered on the same edge. At round r_1 agent a triggers the timeout and goes in state Forward starting moving clockwise. At round r_2 it finds the marked node and terminates. Note that the other agent is either unblocked and thus reach the marked node and correctly terminate or it waits forever on a missing edge.



(b) Phase 2: case of three agents gathered, two on the same node the other on the neighbour. At round r_1 the Retroguard enters in the black hole. At round r_2 the MLeader detects that Retroguard failed to report an terminates. Note that this happens before a timeouts.

Figure 3 Example of runs for Phase 2 of Gather&Locate.

The role of Explorer is to visit new nodes, while Follower just follows Explorer when a node is safe (this is similar to Leader and Avanguard in CautiousPendulum). If the remaining Start agent meets with either the Follower or the Explorer, it will assume the behaviour of the Follower (predicate *meeting*[Follower] in state Init and state Copy). Finally, if the three agents meet on the same node, Phase 1 terminates (see predicate $\#A = 3$ in all states). In all cases, at the end of round $9n$, this phase ends.

In Section 6.2, we will show that, if in Phase 1 all the alive agents have not localised the Bh, then either:

- three agents gathered: either three agents are on the same node, or two agents are on a node v and the third agent is blocked on the clockwise neighbour v' of (the marked) node v; or
- the counter-clockwise neighbour of the black hole has been marked, at most one agent is lost, and the two remaining agents are gathered. The two agents are either on the same node, or on two different neighbours node and one of them has marked the node where the other resides.
- the counter-clockwise neighbour of the black hole has been marked, at most one agent is lost, one agent correctly terminated, while the other is still looking for the Bh.

Algorithm 1 Gather&Locate; Phase 1 – Algorithm for scattered agents – Start.

1: Predicates Shorthands: $NextUnsafe = Etime > EMtime[C]$
2: $NextSafe$ = the agent that marked the node returned.
3: States: {Init, Wait, EndPhase1, Terminate, Copy}.
4: In state Init:
5: CautiousExplore($right$ | $Ttime = 9n \vee \#A = 3$: EndPhase1; *marked*: Wait; *meeting*[Start]: Two; *meeting*[Follower] $\vee$ *meeting*[Explorer]: Copy)
6: In state Wait:
7: Explore(nil | $Ttime = 9n \vee \#A = 3$: EndPhase1; $NextUnsafe$: Terminate; $NextSafe$: Init)
8: In state Two:
9: Use IDs to assign to yourself a role in {Follower, Explorer} in a mutual exclusive fashion.
10: Execute the corresponding Algorithm, that is Alg. 3 in state WaitFollower, or Alg. 2 in state Explorer.
11: In state Copy:
12: set your role to Follower.
13: In state EndPhase1:
14: take the role of Start
15: starts Phase 2 by entering state InitP2 of Alg. 4.
16: In state Terminate:
17: terminate, Bh is the next node in clockwise direction.
18:

Algorithm 2 Gather&Locate; Phase 1 – Algorithm for EXPLORER.

States: {Explorer, Back, MoveForward, EndPhase1, Terminate}. ▷ Terminate and EndPhase1 as in Algorithm 1
In state Explorer:
  if current node is not marked **then**
    mark current node
    EXPLORE($right$ | $Ttime = 9n \vee \#A = 3$: EndPhase1; $Enodes > 0$: Back)
  else
    EXPLORE(nil | $Ttime = 9n \vee \#A = 3$: EndPhase1; $NextUnsafe$: Terminate) ▷ If the node is marked we have to wait to see if it is safe to move
In state Back:
  EXPLORE($left$ | $Ttime = 9n \vee \#A = 3$: EndPhase1; $Enodes > 0$: MoveForward)
In state MoveForward:
  unmark current node
  EXPLORE($right$ | $Ttime = 9n \vee \#A = 3$: EndPhase1; $Enodes > 0$: Explorer)

Algorithm 3 Gather&Locate; Phase 1 – Algorithm for FOLLOWER.

States: {WaitFollower, Follow, EndPhase1}. ▷ EndPhase1 as in Algorithm 1
In state WaitFollower:
  EXPLORE(nil | $Ttime = 9n \vee \#A = 3$: EndPhase1, $Meeting[Back]$: Follow)
In state Follow:
  EXPLORE($right$ | $Ttime = 9n \vee \#A = 3$: EndPhase1, $Enodes > 0$: WaitFollower)

Phase 2. The agents start in InitP2 state of Algorithm 4: here, several checks are executed to understand how Phase 1 ended and to orchestrate the behaviour of the agents. In more details:

- Each agent checks if there are other agents on the same node: in case there are two agents, they get the roles of RETROGUARD and MLEADER (their behaviour is similar to RETROGUARD and LEADER in CautiousPendulum). If there are three agents, they start algorithm CautiousPendulum.
- If the agent is missing the pebble, then it was blocked while trying to recover its pebble at the end of Phase 1. The agent tries to recover the pebble by moving counter-clockwise on one step for $4 \cdot n^2$ rounds. If during this period it succeeds then it goes to state Forward. If $4 \cdot n^2$ rounds have passed without succeeding, then the agent goes in the Forward state. This move has the following goal: if there are two agents on its counter-clockwise node, then they have role RETROGUARD and MLEADER, and in $4 \cdot n^2$ rounds can locate black hole. Otherwise, if there is just one agent on the clockwise node or if there is no one, this timeout avoids that the agent is blocked forever on a missing edge trying to recover a pebble.
- If the agent is on a marked node, then it waits there until either it meets the agent that marked the node, or $4 \cdot n^2$ rounds have passed. If they meet, they get the roles of RETROGUARD and MLEADER; otherwise, if the timeout triggers, the agent goes in state Forward.
- If none of the above applies, the agent goes in state Forward.

We now detail the behaviour of the agents:

- Agent MLEADER and agent RETROGUARD. The MLEADER moves clockwise, while RETROGUARD acts as in Algorithm CautiousPendulum. If RETROGUARD fails to report, MLEADER identifies the black hole and terminates. Finally, if MLEADER and an agent that is not RETROGUARD meet, then this new agent takes the role of AVANGUARD and MLEADER the role of LEADER, and they behave exactly as in Algorithm CautiousPendulum (predicate $meeting[Leader]$ and state BeAvanguard for the agent with role START; and predicate $meeting$[Start] and state GoToCP for the LEADER). The only caveat in this case, is that MLEADER keeps the value of variable $\#Meets[Retroguard]$ when switching to LEADER.

- Agent in state Forward. In state Forward an agent moves in the clockwise direction. If it reaches a marked node, then it discovered the black hole and the agent terminates. If two agents in state Forward meet, they use their IDs to get the roles of MLeader and Retroguard.

Algorithm 4 Gather&Locate; Phase 2 – Algorithm for scattered agents – Start.

Predicates Shorthands: $NextUnsafe = Etime > EMtime[C]$
States: {InitP2, BeAvanguard, Terminate, AssignRoles}.
In state InitP2:
  if $\#A > 1$ **then**
    go to state AssignRoles
  else if my pebble is missing **then**
    Explore($left$ |$meeting$[Start]: AssignRoles; $meeting$[MLeader]: BeAvanguard; $Enodes > 0$: Forward ; $Ttime > 4n^2$: Forward)
  else if the current node is marked **then**
    take the pebble if yours
    Explore(nil |$meeting$[Start]: AssignRoles; $Ttime > 4n^2$: Forward)
  else
    go to state Forward
In state Forward:
  Explore($right$ | $marked \wedge Enodes > 0$: Terminate; $meeting$[Start]: AssignRoles; $meeting$[MLeader]: BeAvanguard)
In state AssignRoles:
  if your pebble is on the node take it.
  if #A=2 **then**
    Use ID to take a role in { Retroguard, MLeader } in a mutual exclusive fashion.
    execute the CautiousPendulum if Retroguard or Alg. 5 in state Go if MLeader.
  else
    Use ID to take a role in { Retroguard, Leader, Avanguard } in a mutual exclusive fashion.
    start algorithm CautiousPendulum.
In state BeAvanguard:
  if your pebble is on the node take it.
  take the role of Avanguard.
  start algorithm CautiousPendulum.
In state Terminate:
  Terminate Bh is the next node in clockwise direction.

Algorithm 5 Gather&Locate; Phase 2 – Algorithm for MLeader.

Predicates Shorthands: $NextUnsafe = Etime > EMtime[C]$.
$FailedReport[Retroguard] = EMtime[C] > 2((\#Meets[Retroguard] + 1) + Tnodes)$.
States: {Go, Cautious, StartCP, Terminate, TerminateR}.
In state Go:
  Explore($right$ | $marked$: Cautious; $meeting$[Start]=StartCP; $FailedReport[Retroguard]$: TerminateR;)
In state Cautious:
  Explore(nil | $meeting$[Start]: StartCP; $NextUnsafe$: Terminate; $FailedReport[Retroguard]$: TerminateR)
In state StartCP:
  start algorithm CautiousPendulum with the role of Leader keeping the value of variable $\#Meets[Retroguard]$.
In state Terminate:
  Terminate, Bh is in the next node in clockwise direction.
In state TerminateR:
  Terminate, Bh is in the node that is at distance $\#Meets[Retroguard] + 1$ from counter-clockwise direction from the reference node.

6.2 Correctness of Gather&Locate.

▶ **Definition 8** (Gathered configuration). *We say that a group of k agents* gathered *if either:*

- *There are k agents on the same node; or,*
- *There are $k-1$ agents on node v_i, and one agent a on node v_{i+1}. Moreover, agent a marked node v_i with a pebble and has to still unmark it.*

Let us first start with a technical lemma, derived from [12], and adapted to our specific case.

▶ **Lemma 9.** *If k agents perform a cautious walk in the same direction for an interval I of $9n$ rounds and one of the alive agents does not explore n nodes and no agent terminates, then the agents gathered.*

Proof. Let A be the set of agents performing a cautious walk, say in clockwise direction, and let a^* be the agent that does not explore n nodes.

Agent a^* can be blocked in progressing its cautious walk in two possible ways: (i) when it is trying to explore a new node using a missing edge in its clockwise direction (we say that a^* is *forward blocked*); (ii) when it is returning to a previously explored node to unmark it (it is blocked by an edge missing in its counter-clockwise direction, and we say that a^* is *backward blocked*). Thus, If in a round r an agent is forward blocked and another one is backward blocked, then they are on two endpoints of the same missing edge.

If a^* is not blocked for $3(n-1)$ rounds then it has explored n nodes. Therefore, a^* has been blocked for at least $6n-3$ rounds or more rounds over an interval of $9n$ rounds. If there is a round r' when a^* is blocked, then every $a \in A$ that at round r' is not blocked does move (note that all blocked agents are either backward or forward blocked on the same edge of a^*).

Thus, all agents in A that are not blocked move towards a^* of at least $\frac{6n-3}{3} = 2n-1$ steps. On the other hand, every time a^* moves, the other agents might be blocked; however, by hypothesis, this can occur less than $3n$ times.

Since the initial distance between a^* and an agent in A is at most $n-1$, it follows that such a distance increases less than $n-1$ (due to a^* movements); however, it decreases by $2n-1$ (due to a^* being blocked). In conclusion, by the end of I, all agents are either at the same node or at the two endpoints of a missing edge and the lemma follows. ◀

▶ **Lemma 10.** *Given three agents executing Phase 1, at most one of them enters the black hole. In this case, the counter-clockwise neighbour node of the black hole is marked by a pebble.*

Proof. If agents have not already met, then each agent performs a cautious walk, all in the same direction, marking a node and avoiding that other agents visit a possibly unsafe node (see state Init in Algorithm 1): when the agent sees a marked node, it goes in state Waits. In this state, the agent waits until it is sure that the next node is safe (that is, until the agent that marked the node returns to remove the pebble).

When two agents meet, they become Follower and Explorer. By construction, Follower never reaches Bh: in fact, Follower moves a step clockwise only when it sees Explorer returning (see state Wait and predicate *meeting*[*Explorer*]); this implies that the node where it moves is safe. Also note that Explorer never visits a possibly unsafe node if there is another agent on it: in fact, in state Explore, there is a check on whether the current node is marked or not; if marked, Explorer waits (thus, also blocking Follower) until the next node can be deemed as safe.

If the third agent reaches Follower, it will also become Follower and it will never visit an unsafe node. Moreover, the Explorer agent always marks a node before visiting its unexplored neighbour (see state Explore of Algorithm 2).

In conclusion, we have that at most one agent enters Bh, and the counter-clockwise neighbour node of Bh will be marked by a pebble, and the lemma follows. ◀

▶ **Observation 11.** *If an agent terminates while executing Phase 1, then it correctly terminates.*

Proof. The claim follows immediately by observing that the state Terminate is always reached when an agent visits a marked node, the clockwise edge is not missing, and the agent that marked the node does not return. ◀

▶ **Lemma 12.** *Let us consider three agents executing Phase 1. If not all agents terminated locating the* Bh*, then Phase 1 ends by round* $9n$ *and, when it ends, one of the following scenarios holds:*

- *(1) all agents gathered;*
- *(2) at most one agent disappeared in the black hole, the counter-clockwise neighbour of the black hole is marked, and the remaining agents gathered;*
- *(3) one agent terminated, the counter-clockwise neighbour of the black hole is marked, and the remaining agent has to still locate the* Bh*.*

Proof. By construction, in all states the agents check predicate $Ttime = 9n$; thus, Phase 1 ends after at most $9n$ rounds. By Lemma 10 we have that at most one agent enters the Bh, leaving its counter-clockwise neighbour marked. There are three possible cases:

- One agent terminates, and by Observation 11 it terminates correctly solving the BHS-Problem. The other agent has to still locate the Bh
- One agent enters in the Bh and no one terminates. If no alive agent terminates, then no one of them has explored n nodes. Therefore, at the end of Phase 1 $Ttime = 9n$ and by Lemma 9 the agents gathered, and the lemma follows.
- No agent enters the Bh and no agent terminates. In this case we have that three agents gather by the end of Phase 1. If agents end Phase 1 because predicate $\#A = 3$, then the statement immediately follows. Otherwise, $Ttime = 9n$, by Lemma 9 the agents gathered, and the lemma follows. ◀

The next lemma shows that, if Bh has been marked in Phase 1, then two agents executing Algorithm 4 solve BHS-Problem in at most $\mathcal{O}(n^2)$ rounds.

▶ **Lemma 13.** *Let us assume that the counter-clockwise neighbour* v *of* Bh *has been marked by a pebble. If two agents execute Algorithm 4, at least one of them terminates correctly locating the* Bh *in* $\mathcal{O}(n^2)$ *rounds; the other agent either terminates correctly locating the* Bh *or it never terminates.*

Proof. By Lemma 12, at the first round of Phase 2 we have two possible cases:

- The two agents are at the same node. In this case, they immediately enter in state AssignRoles. Let a be the agent that takes the role of MLeader and b be the one that becomes Retroguard. Their movements are similar to the ones of Leader and Retroguard in CautiousPendulum, with the only difference that MLeader moves until it reaches a marked node. By Lemma 12, this marked node is the counter-clockwise neighbour of Bh; thus, if MLeader reaches it, MLeader correctly terminates.
  If MLeader does not visit the marked node because of a missing edge, Retroguard is able to move. By using a similar argument to the one used in the proof of Theorem 5, the black hole is located in at most $\mathcal{O}(n^2)$ rounds, and the lemma follows. Also note that the only agent that can go in a termination state is MLeader, therefore Retroguard cannot terminate incorrectly.
- The two agents occupy two neighbouring nodes, and the most clockwise agent does not have the pebble. More precisely, agent a is at node v, agent b at node v'; also, agent b is missing its pebble, and node v is marked by a pebble. In this case, agent a executes lines 9-10 of Algorithm 4: it removes the pebble from v, and waits for $4 \cdot n^2$ rounds. Agent b executes line 7 of Algorithm 4: it moves towards node v for $4 \cdot n^2$ rounds.

If edge $e = (v, v')$ appears before the timeout, then a and b meet, and previous case applies. Otherwise, both agents go in state Forward. In this state they both move clockwise. If one of them reaches the marked node, it correctly terminates. Otherwise the path towards BH is blocked by a missing edge and the agents would meet in at most $\mathcal{O}(n)$ rounds. When they meet, they both go in state AssignRoles, and previous case applies again. Note that this implies that at most one of the agents in state Forward can be blocked forever by a missing edge. ◀

▶ **Lemma 14.** *Let us assume that three agents are gathered after Phase 1. Then, if three agents executes Algorithm 4, at least one of them terminates correctly locating the* BH *in* $\mathcal{O}(n^2)$ *rounds; the other agents either terminate correctly locating the* BH *or never terminate.*

Proof. If three agents are on the same node, then they start CautiousPendulum algorithm and the correctness follows from Theorem 5. Otherwise, we have two agents on a node v, with v marked with a pebble, and the other agent b on v', with v' the clockwise neighbour of v. Upon the start of Phase 2, the two agents will become RETROGUARD and MLEADER, respectively; MLEADER waits on the marked node, while b tries to go back to v.

If edge $e = (v, v')$ is missing for $4n^2$ rounds, then RETROGUARD has enough time to reach the black hole, and MLEADER to terminate because of the fail to report of RETROGUARD, hence the lemma follows. Note that after the termination of MLEADER, the agent b goes in state Forward, it moves clockwise and either it enters the BH or is blocked forever by a missing edge; in all cases it cannot terminate.

Finally, the last case to analyse is when RETROGUARD is blocked by a missing edge in the first $4n^2$ rounds. In this case, MLEADER and b meet, and algorithm CautiousPendulum starts. The lemma follows by Theorem 5. ◀

▶ **Lemma 15.** *Let us assume that a single agent a starts Phase 2. This agent, by executing Algorithm 4, either terminates correctly or it waits forever on a missing edge.*

Proof. By algorithm construction after at most $4n^2$ rounds from the beginning of Phase 2 the agent a goes in state Forward and it starts moving in clockwise direction. Being the only agent still active, it will never change behaviour until it reaches the marked node or another agent.

By Lemma 12 the counter-clockwise neighbour of BH is marked, and the terminated agent is located at that node. If no edge is removed forever, a will reach the marked node, and it will terminate. Otherwise, a will be forever blocked on a missing edge. ◀

▶ **Theorem 16.** *Given a dynamic ring $\mathcal{R}$, three agents with visible IDs and pebbles running* Gather&Locate*, solve* BHS *in* $\mathcal{O}(n^2)$ *moves and* $\mathcal{O}(n^2)$ *rounds.*

Proof. By Lemma 12, Phase 1 terminates in at most $\mathcal{O}(n)$ rounds. At this time, either: (1) BHS-PROBLEM is solved and all agents terminated, or (2) the agents gathered, or (3) the counter-clockwise neigbhour of BH is marked and the remaining agents are gathered, or (4) there is still an agent active while an agent correctly terminated. In case (2), the proof follows by Lemma 14. In case (3), the proof follows by Lemma 13. In case (4), the proof follows by Lemma 15. ◀

By Th. 7 and Th. 16 we have:

▶ **Theorem 17.** *Algorithm* Gather&Locate *is size-optimal with optimal cost and time.*

7 Conclusion

In this paper, we have investigated the Bhs on dynamic rings in scenarios where agents are scattered. However, several questions remain open. In particular, we believe that the findings in Observation 6 could potentially be applied to any constant number of agents. Another unresolved issue is the examination of unoriented rings with scattered and anonymous agents, as well as the development of faster Bhs algorithm for groups larger than three agents.

References

1 S. Abshoff and F. Meyer auf der Heide. Continuous aggregation in dynamic ad-hoc networks. In *21st Int. Coll. on Structural Inf. and Comm. Compl.*, pages 194–209, 2014. doi:10.1007/978-3-319-09620-9_16.

2 B. Balamohan, P. Flocchini, A. Miri, and N. Santoro. Time optimal algorithms for black hole search in rings. In *10th International Conference on Combinatorial Optimization and Applications*, pages 58–71, 2010. doi:10.1007/978-3-642-17461-2_5.

3 M. Bournat, A.K. Datta, and S. Dubois. Self–stabilizing robots in highly dynamic environments. In *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 54–69, 2016. doi:10.1007/978-3-319-49259-9_5.

4 M. Bournat, S. Dubois, and F. Petit. Computability of perpetual exploration in highly dynamic rings. In *37th IEEE International Conference on Distributed Computing Systems*, pages 794–804, 2017. doi:10.1109/ICDCS.2017.80.

5 A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *Int. J. Parallel Emergent Distributed Syst.*, 27(5):387–408, 2012. doi:10.1080/17445760.2012.668546.

6 J. Chalopin, S. Das, A. Labourel, and E. Markou. Tight bounds for black hole search with scattered agents in a synchronous ring. *Theoretical Computer Science*, 509:70–85, 2013. doi:10.1016/J.TCS.2013.02.010.

7 J. Czyzowicz, D. Kowalski, E. Markou, and A. Pelc. Complexity of searching for a black hole. *Fundamenta Informaticae*, 71:229–242, 2006. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-05.

8 J. Czyzowicz, D. Kowalski, E. Markou, and A. Pelc. Searching for a black hole in synchronous tree networks. *Combinatorial Probabilistic Computing*, 16(4):595–619, 2007. doi:10.1017/S0963548306008133.

9 G.A. Di Luna. *Mobile Agents on Dynamic Graphs,* Chapter 20 of [20]. Springer, 2019.

10 G.A. Di Luna and R. Baldoni. Brief announcement: Investigating the cost of anonymity on dynamic networks. In *34th Symposium on Principles of Distributed Computing*, pages 339–341, 2015. doi:10.1145/2767386.2767442.

11 G.A. Di Luna, S. Dobrev, P. Flocchini, and N. Santoro. Distributed exploration of dynamic rings. *Distributed Computing*, 33:41–67, 2020. doi:10.1007/S00446-018-0339-1.

12 G.A. Di Luna, P. Flocchini, L. Pagli, G. Prencipe, N. Santoro, and G. Viglietta. Gathering in dynamic rings. *Theoretical Computer Science*, 811:79–98, 2020. doi:10.1016/J.TCS.2018.10.018.

13 G.A. Di Luna, P. Flocchini, G. Prencipe, and N. Santoro. Black hole search in dynamic rings. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 987–997, 2021. doi:10.1109/ICDCS51616.2021.00098.

14 S. Dobrev, P. Flocchini, R. Královič, and N. Santoro. Exploring an unknown dangerous graph using tokens. *Theoretical Computer Science*, 472:28–45, 2013. doi:10.1016/J.TCS.2012.11.022.

15 S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Searching for a black hole in arbitrary networks: optimal mobile agents protocols. *Distributed Computing*, 19(1):1–35, 2006. doi:10.1007/S00446-006-0154-Y.

16 S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile search for a black hole in an anonymous ring. *Algorithmica*, 48(1):67–90, 2007. doi:10.1007/S00453-006-1232-Z.

17 S. Dobrev, N. Santoro, and W. Shi. Locating a black hole in an un–oriented ring using tokens: The case of scattered agents. In *13th International Euro–Par Conference European Conference on Parallel and Distributed Computing*, pages 608–617. Springer, 2007. doi:10.1007/978-3-540-74466-5_64.

18 P. Flocchini, D. Ilcinkas, and N. Santoro. Ping pong in dangerous graphs: optimal black hole search with pebbles. *Algorithmica*, 62(3-4):1006–1033, 2012. doi:10.1007/S00453-011-9496-3.

19 P. Flocchini, M. Kellett, P. Mason, and N. Santoro. Searching for black holes in subways. *Theory of Computing Systems*, 50(1):158–184, 2012. doi:10.1007/S00224-011-9341-8.

20 P. Flocchini, G. Prencipe, and N. Santoro *(Eds.). Distributed Computing by Mobile Entities*. Springer, 2019. doi:10.1007/978-3-030-11072-7.

21 B. Haeupler and F. Kuhn. Lower bounds on information dissemination in dynamic networks. In *26th Int. Symp. on Distributed Computing*, pages 166–180, 2012. doi:10.1007/978-3-642-33651-5_12.

22 F. Kuhn, T. Locher, and R. Oshman. Gradient clock synchronization in dynamic networks. *Theory of Computing Systems*, 49(4):781–816, 2011. doi:10.1007/S00224-011-9348-1.

23 F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *42nd ACM Symposium on Theory of Computing*, pages 513–522, 2010. doi:10.1145/1806689.1806760.

24 E. Markou and M. Paquette. Black hole search and exploration in unoriented tori with synchronous scattered finite automata. In *14th International Conference on Principles of Distributed Systems*, pages 239–253, 2012. doi:10.1007/978-3-642-35476-2_17.

25 E. Markou and W. Shi. *Dangerous Graphs,* Chapter 18 of [20]. Springer, 2019.

26 R. O'Dell and R.Wattenhofer. Information dissemination in highly dynamic graphs. In *Joint Workshop on Foundations of Mobile Computing*, pages 104–110, 2005. doi:10.1145/1080810.1080828.

Sketching the Path to Efficiency: Lightweight Learned Cache Replacement

Rana Shahout[1]
Harvard University, Cambrdige, MA, USA

Roy Friedman
Technion, Haifa, Israel

Abstract

Cache management policies are responsible for selecting the items that should be kept in the cache, and are therefore a fundamental design choice for obtaining an effective caching solution. Heuristic approaches have been used to identify access patterns that affect cache management decisions. However, their behavior is inconsistent, as they can perform well for certain access patterns and poorly for others. Given machine learning's (ML) remarkable achievements in predicting diverse problems, ML techniques can be applied to create a cache management policy. Yet a significant challenge arises from the memory overhead associated with ML components. These components retain per item information and must be invoked on each access, contradicting the goal of minimizing the cache's resource signature.

In this work, we propose *ALPS*, a light-weight cache management policy that takes into account the cost of the ML component. *ALPS* combines ML with traditional heuristic-based approaches and facilitates learning by identifying several statistical features derived from space-efficient *sketches*. ALPS's ML process derives its features from these sketches, resulting in a lightweight and highly effective meta-policy for cache management. We evaluate our approach over real-world workloads run against five popular heuristic cache management policies as well as a state-of-the-art ML-based policy. In our experiments, ALPS always obtained the best hit ratio. Specifically, ALPS improves the hit ratio compared to LRU by up to 20%, Hyperbolic by up to 31%, ARC by up to 9% and W-TinyLFU by up to 26% on various real-world workloads. Its resource requirements are orders of magnitude lower than previous ML-based approaches.

2012 ACM Subject Classification Information systems → Data streams

Keywords and phrases Data streams, Memory Management, Cache Policy, ML

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.34

Supplementary Material *Software (Source Code)*: `https://github.com/r4n4sh/sketch_learning` archived at `swh:1:dir:bcbd252fd90d7f6377a0443811576d59ca79e1fd`

Funding *Rana Shahout*: Supported in part by Schmidt Futures Initiative and Zuckerman Institute. *Roy Friedman*: Israel Science Foundation grant #3119/21.

Acknowledgements We thank Ohad Eytan for helping run Caffeine's simulator.

1 Introduction

Caching is a fundamental performance boosting technique, widely used by middleware, operating systems, databases, data-stores, edge servers, and content delivery networks [10, 14, 15, 21, 23, 34, 19, 17, 35, 12]. A cache improves the system's average response time by storing certain items closer to their consumers. This way, future accesses to cached items are served faster than serving them from their main storage. Caching of responses can also save communication and computations needed to re-calculate remote invocations.

[1] Corresponding author

Alas, caches usually cannot hold all accessed items, meaning that a *cache management policy* is needed to decide which items should be stored in the cache. An access to a cached item is called a *hit*; otherwise it is a *miss*. In particular, the cache management policy must predict what items are likely to be accessed in the future, in order to maximize the ratio of hits to all accesses (*hit ratio*), thereby minimizing the expected access latency of the entire system.

Traditionally, caching relies on heuristic approaches to identify patterns and statistically significant signals within the access history. A notable example is LRU [25], which operates under the assumption that the recency of an item's last access serves as a reliable indicator of its future access likelihood. LRU incorporates newly arrived items into the cache while evicting the least recently accessed item when the cache reaches its capacity limit.

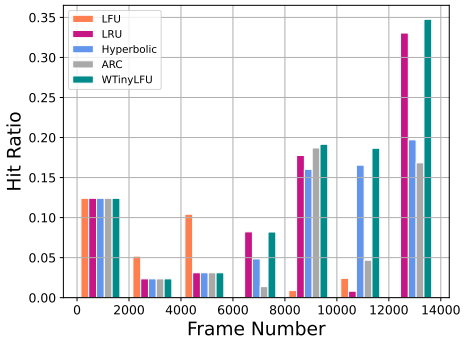
Frequency is another important signal, at least for certain workloads. The respective LFU policy [2] and its variants [13, 29, 3, 2, 20, 21] prioritize items based on their access frequency. To accommodate dynamic changes in item popularity over time, some variants incorporate aging or freshness mechanisms. However, predictions based solely on recency or frequency may not always yield good cache performance. The inclusion of inter-access or inter-reference, which captures temporal item access relationships, has shown promise in improving predictions. Further, it is now a common belief that adaptive combinations of recency, inter-reference, and frequency yield superior cache performance across a wide range of workloads [28, 39, 37, 33, 9, 20].

Given the success of machine learning (ML) in prediction tasks, we may apply ML for cache management. This involves training models to predict which items are likely to have the lowest hit rate and therefore should be prime candidates for eviction [47, 5]. However, a drawback of these ML-based approaches is the associated memory overhead. These methods store information per item and require invoking ML mechanisms for every access, resulting in substantial memory and computational requirements. Alternatively, CACHEUS [43] utilizes ML to decide which among a few cache management experts to use at any given moment. Yet, CACHEUS invokes ML on every access, and its learning and decisions are based directly on the access history.

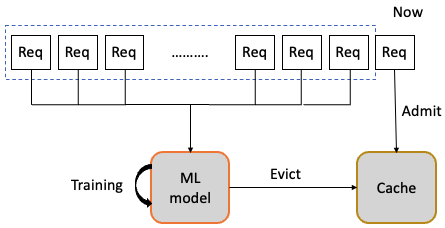
Since caches are meant to serve as performance optimizations, one aspires to keep the cache's resource signature as low as possible. Also, learning and predicting are based on the raw access stream, which lengthens the learning process and makes predictions expensive.

Motivation. When analyzing the hit ratio performance of various widely deployed caching algorithms across a comprehensive set of workloads, we see that there is no silver bullet across all workloads. The effectiveness of different policies varies depending on workload characteristics and cache size. Moreover, as the workload evolves over time, the most suitable heuristic caching algorithm for the same workload may change. To investigate this dynamic behavior, we partition the access sequence into frames and assess the performance of different caching techniques within each frame. Figure 1a illustrates this analysis using the Mergep trace, where each frame consists of 10K accesses. It is shown that the winning heuristic approach changes as the trace progresses, but no single policy dominates across all frames. Notably (not shown here due to lack of space), subsequent frames often exhibit the same winning scheme. In summary, despite numerous attempts to identify an ideal caching policy, state-of-the-art algorithms demonstrate inconsistent performance across diverse workloads and even within the same workload, primarily due to their inherent variability.

The application of ML to cache management has been limited, despite its wide application across several domains. A previous approach, as illustrated in Figure 1b, involved utilizing ML to derive new caching policies. When the cache reaches full capacity and an item is accessed,



(a) Hit ratio.



(b) Existing ML based policies.

Figure 1 (a) Hit ratio of different heuristic approaches as a function of the frame number on sampled frames using the Mergep trace with a cache size of 100K items (b) General architecture of existing ML based policies. The ML model is invoked on each access to advise which item to evict.

Table 1 The allocated space of the metadata for LRB in different traces (values taken from [47]).

Cache size	Wiki	A1	A2
128 GB	2.68 GB	0.76 GB	1.79 GB
256 GB	3.5 GB	1.28 GB	3.07 GB
512 GB	5.12 GB	2.04 GB	5.12 GB
1 TB	6 GB	3 GB	7 GB

this approach leverages the ML model to make eviction decisions. However, it faces two primary issues. First, storing **per item** past information within a sliding memory window for training and prediction incurs significant memory overhead. Second, the prediction overhead associated with each eviction operation further adds to the computational burden. To quantify the memory overhead, Table 1 presents the metadata requirements for LRB [47]. The memory overhead is contingent upon the cache size and the specific trace, and it increases as the cache size grows larger.

Our approach. To reduce ML costs, ALPS divides the access stream into frames and applies ML to predict the most suitable heuristic caching policy for the subsequent frame. By doing so, we reduce computational overhead by executing ML only once per frame. This is significantly more efficient than applying ML on every cache access. Further, to maintain per-frame information memory efficiently, ALPS utilizes memory frugal sketches. These sketches serve as concise and structured representations of previously identified heuristic caching patterns. At the end of each frame, the condensed and structured information is fed into ALPS, which applies the predicted most appropriate policy to the next frame. This approach combines the benefits of known heuristic approaches with ML in an efficient manner. We highlight that ALPS learns from sketches' output rather than the raw access stream, which reduces memory overhead. The sketches employed in ALPS include HLL [22] for estimating the number of unique items accessed in a frame, Count-Min sketch [16] for estimating item frequencies and generating a frequency histogram, and the Space Saving algorithm [38] to identify heavy hitters. Finally, we developed a novel sketch called ISketch (Inter-Reference Sketch), which captures the inter-reference data of the most frequently accessed items within each frame and is fed into ALPS. We compared ALPS with established heuristic- and ML-based cache

management policies. Our analysis reveals that ALPS outperforms these approaches by achieving a higher hit ratio while significantly reducing memory overhead compared to LRB [47] and CACHEUS [43].

Contributions. We make the following contributions: (1) We design ALPS, a generic caching framework that achieves a high hit ratio while reducing memory overheads compared to prior ML-based approaches. (2) As a building step, we present ISketch, a novel algorithm for tracking inter-arrival times, which allows us to track recency during each frame. (3) We evaluate ALPS against well-known heuristic based cache algorithms and learning based algorithms. ALPS exceeds most efficient heuristic-based cache policies' hit rates, improving LRU by up to 20%, Hyperbolic by up to 31%, ARC by up to 9% and W-TinyLFU by up to 26% depending on workloads. In addition, ALPS decreases the memory overhead by an order of magnitude compared to the state-of-the-art ML-based approaches while improving the hit ratio by $6\% - 9\%$ compared to LRB and $7.5\% - 12\%$ compared to CACHEUS. (4) ALPS achieves a low training and inference overhead by extracting the most effective features from highly memory-efficient sketches and applying ML only once per frame.

To summarize our key insights, we argue that: (1) combining existing cache management policies in a smart way is likely to yield better improvements than inventing another one, (2) succinct sketches can capture data access patterns and serve as effective features for ML, (3) operating at time frame granularity rather than single access granularity yields more efficient ML based solutions.

2 Background and Related Work

2.1 Caching Algorithms

Least Frequently Used (LFU) [11, 44] aims at maintaining the most frequently used items in the cache. To that end, when the cache is full, LFU removes the item with the lowest reference frequency from the cache. This is done by tracking how many times each item is referenced in the cache.

Yet, in most practical workloads, the access frequency radically changes over time. Hence, there is no point in keeping an item in the cache once its popularity has faded, just because it was once very popular. As a result, LFU variants include aging algorithms or focus on a small window of the last W accesses alone, as done, e.g., in Window LFU (WLFU) [29].

Least Recently Used (LRU) [25] always inserts the last accessed item into the cache, and the Least Recently Used item is evicted when the cache is full. LRU adapts automatically to variations in data access patterns. Practical systems often only realize an approximation of LRU, e.g., through sampling [42] or Clock [4, 26] to reduce execution overheads and eliminate concurrency hot-spots. Segmented LRU (SLRU) [30] distinguishes between items that are temporarily popular and are accessed twice or more in a short period of time and those that are accessed just once during that period. LRU-K [39] combines concepts from LRU and LFU. 2Q [28] is an effective practical approximation of LRU-K.

Hyperbolic [9] is a recent proposal for combining frequency with recency in a holistic manner. With the same internal data structures, hyperbolic caching can act like a number of different eviction policies, changing its behavior based on the workload. The main idea is that an item's temporal priority is set to its frequency since it was last inserted into the cache, divided by the time that it spent in the cache, multiplied by a parameterized factor. The cache victim is the item with the lowest temporal priority.

Adaptive Replacement Cache (ARC) [37] is an adaptive caching algorithm that takes both recency and frequency of accesses into account. The cache is divided into two LRU lists, T1 and T2. T1 contains items that have been accessed only once, whereas T2 contains items that have been accessed multiple times after admission. In addition, ARC maintains ghost entries for both T1 and T2, which aid in deciding how to dynamically adapt the relative sizes of T1 and T2. Due to the fact that ARC uses an LRU list for T2, it is not possible to get the full frequency distribution of the workloads and perform well under LFU-friendly workloads.

Adaptive W-TinyLFU is the management policy of the Caffeine Java 8 cache [36], Go based Ristretto [18], and Rust based Moka [31]. W-TinyLFU has three parts: the Main cache, TinyLFU – an approximated LFU based admission filter, and a Window cache. New items are added to the Window cache; it can be kept using any known policy, but all known implementations employ LRU. The Main cache can use any cache management strategy, although known realizations employ SLRU. The filter utilizes a space-efficient sketch [16] to approximately track the access frequencies of a large number of items, well beyond the cache size. Whenever an item is evicted from the window cache, it is compared by the TinyLFU filter to the would be main cache victim; the item with the highest estimated frequency gets to be in the main cache and the other is deleted.

Least Hit Density (LHD) [5] is based on hit density, a workload-agnostic metric for ranking objects during eviction. It monitors objects online and uses conditional probability to predict their likely behavior. LHD predicts the expected number of hits per space unit consumed by each object (hit density) and eliminates objects that contribute little to the cache's hit rate. LHD does not rely on heuristics but rather rigorously models objects' behavior using conditional probability to modify its behavior in real time.

Learning Relaxed Belady (LRB) [47] approximates a variant of Belady's algorithm [7], called Belady's MIN (oracle) algorithm, using ML to find objects to evict based on past access patterns. To reduce the cost of ML, the authors first came up with a relaxed Belady algorithm that evicts an object whose next request is above a certain threshold but not necessarily the farthest in the future. For this, LRB keeps information **about an object** within a defined sliding memory window. The information within the sliding memory window is used for training and prediction. Thus, LRB invokes ML on each access, and the predictions are invoked once the full cache has to evict an object.

CACHEUS [43] identifies several cache management experts (ARC, LIRS, LFU, SR-LRU, and CR-LFU) and uses ML to predict which one to use at any given time based on workload primitive types. This is based on classifying workload primitives into: LRU-friendly, LFU-friendly, scan, and churn. CACHEUS uses online reinforcement learning with regret minimization to provide a caching method that aims to optimize for dynamically manifesting workload primitive types. CACHEUS invokes ML on each access and bases its learning and prediction decisions directly on the access history.

MiniSim [51] is a generic framework that uses multiple scaled-down simulations to explore candidate cache configurations simultaneously. It consists of multiple shadow caches, each activating a different management policy for a sample of the workload. Periodically, MiniSim checks which shadow cache works best, and switches the real cache to the policy that performed best among the shadow caches during the last such period. Additionally, MiniSim

Table 2 Comparison of the sketches that appear in the paper. ϵ is the estimation accuracy parameter.

Algorithm	Space	Update Time	Randomization	Comments
SS [38]	$O(\epsilon^{-1})$	$O(1)$	Deterministic	implemented according to [8]
HLL [22]	$O((\log\log D)/\epsilon^2)$	$O(1)$	Randomized	D is distinct elements number
Count-Min [16]	$O(\epsilon^{-1}\log\delta^{-1})$	$O(\log\delta^{-1})$	Randomized	δ is the probability of failure
ISketch	$O(k)$	$O(1)$	Deterministic	k is the number of entries

performs a Talus-like [6] performance cliff removal transparently for complex policies. The main shortcoming of MiniSim is that to be memory and computationally viable, the sampling probability needs to be very small, which misses out on certain phenomena, resulting in sub-optimal behavior [21].

2.2 Sketches

A sketch is a space efficient data structure that provides a fast data synopsis of the dataset, often sacrificing accuracy for space frugality. The tradeoffs one has to consider in designing sketches are accuracy error, space consumption, and processing time complexity. Table 2 lists the analytical performance summary of several algorithms mentioned below.

Space Saving (SS) [38] finds the most frequently occurring elements in a data stream, a.k.a., *heavy hitters*. SS processes a stream of identifiers in order to determine their frequency. It keeps track of a collection of $\frac{1}{\epsilon}$ integer counters, each with its own unique item ID. When a new item is received, SS increments its counter, if it has one. Otherwise, SS gives the item a minimal-valued counter before incrementing it (disassociating the previous ID). As an example, suppose the smallest counter is associated with ID x and has a value of 4; if y comes and does not have a counter, it will take over x's counter and increment it to 5 (thereby leaving x without a counter). When an item's frequency is queried, SS returns the value of its counter if it has one, or the value of the minimal counter otherwise. Suppose the total number of insertions handled by SS is Z, then the sum of counters equals Z and hence the minimal counter is at most $Z\epsilon$. Hence, SS frequency estimates have a maximum error of $Z\epsilon$.

Count-Min Sketch [16] can be used to estimate items' frequencies over a stream without explicitly remembering any identifiers. It consists of a set of d independent hash functions $\{h_j()|j \in [1,\ldots,d]\}$ and a two-dimensional array of counters of width w and depth d. To add an item x with a value of v_x, we increment the counters at $CM[j,h_j(x)]$ by v_x for $1 \le j \le d$.

A query on an item is returns the minimum of the respective counters. For a stream of size N, CM sketch guarantees that its frequency estimation is correct up to an additive $N \cdot \epsilon$-error with a probability of at least $1-\delta$ where $d = \log\delta^{-1}$.

HyperLogLog (HLL) [22] is a probabilistic data structure that counts the number of distinct elements in a multiset. HLL applies a hash function $h()$ to the identifier of each element, and remembers the maximal number of leading zeros in all hash values. The more leading zeros there are, the higher the cardinality is. If the bit pattern 0^{L-1} is at the beginning of the remembered value, a good estimate for the size of the multiset is 2^L. HLL guarantees a relative accuracy of $1.04/\sqrt{m}$ given a memory budget of m units.

3 ALPS DESIGN

This section presents the design of Adaptable Learned Policy Selection (ALPS), a framework that utilizes machine learning (ML) to dynamically switch between a number of cache management algorithms. The core objective of ALPS is to address the cache replacement problem by predicting the optimal heuristic-based algorithm and configuring the cache management policy for the subsequent time frame.

Formally, the prediction problem can be defined as follows: given the latest W accesses, the task is to predict the cache management policy that maximizes the hit ratio in the next W arrivals, where W represents the frame size. This prediction-based approach offers several advantages. Firstly, it consistently outperforms heuristic-based methods in cache hit ratio. Secondly, since the prediction algorithm is executed once every W accesses, the inference time is short, and the associated cost is amortized over W accesses.

ALPS structure. Figure 2 illustrates ALPS's workflow. It comprises two independent data structures: the *Sketches component* (Section 3.1) and the *ML submodel* (Section 3.3). Unlike previous ML-based caching approaches, ALPS divides the access sequence into fixed-sized intervals called frames to facilitate its operation. The Sketches component consists of four sketches, namely SS, HLL, CM, and ISketch (a novel sketch introduced in this work to track inter-arrival times), as discussed in Section 3.1. With each item's access, these sketches capture relevant statistical indicators related to recency and frequency bias within the workload during the last frame, as depicted in Figure 2. At the end of each frame, the statistical indicators extracted from the sketches are used as input features for the ML mechanism. Following that, the sketches are flushed, and all their counters are reset.

To train the ML submodel, we perform supervised learning offline using real-world traces. Ground truths are derived using an established caching simulator [36]. To minimize system overhead, effective features are extracted from the data and utilized as input for the trained model.

A straightforward implementation involves maintaining only the metadata required by all policies and only maintaining a single copy of the data itself for the winning policy. This approach is feasible as metadata is typically much smaller than the actual data, often requiring only a counter or pointer per item. Hyperbolic, for instance, necessitates two counters, but they can be repurposed from LFU's frequency counter and a recency timestamp from LRU. Switching policies at the end of a frame simply involves changing the metadata version and employing a different replacement algorithm for eviction decisions. More sophisticated optimizations are left for future work.

ALPS Update Workflow. Each access to the cache triggers updates to all sketches within the Sketches component. According to Table 2, the update operation for HLL has a constant time complexity of $O(1)$. SpaceSaving (and ISketch (Section 3.2)) can be implemented with a linked list data structure by keeping items with equal counts in a group, resulting in an $O(1)$ update time. The $O(1)$ time complexity means the algorithm consistently executes a small number of operations, rather than requiring thousands. On the other hand, the update operation for CM has a time complexity of $O(\log \delta^{-1})$, where δ, which is a tunable parameter, represents the configuration parameter for CM. By performing these updates, ALPS effectively maintains **per frame** information in the form of *statistical indicators* (Section 3.1). This approach differs from previous learning-based approaches that stored **per item** information.

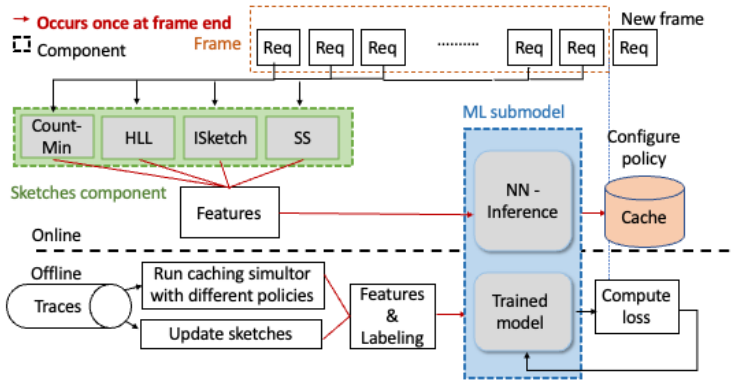


Figure 2 Detailed architecture overview of ALPS.

3.1 Features and Sketches Component

Recency and *frequency* are two prominent signals utilized for designing cache management policies. Recency denotes the time that has elapsed since an item's last access. It provides insight into the likelihood of future accesses based on locality principles. Empirically, there is often a strong correlation between item access frequency and the probability of future access. Additionally, the number of unique items within a single frame offers a valuable indication of access distribution skew. Our objective is to identify the most informative features while maintaining high accuracy and imposing minimal space overhead. To assess the recency versus frequency bias within a workload frame, we define a set of *statistical indicators* as features for the ML submodel. In the context of ALPS, the following indicators are identified:

Number of most accessed items: refers to the count of heavy-hitters in the workload frame.

Maximum frquency: the maximal frequency count.

Average frequency: the average frequency among the most accessed items.

Unique count: the number of distinct items in the frame.

Frequency distribution: this vector captures the distribution of frequencies, with each entry i representing the count of occurrences for frequencies with a value of i.

Minimum inter-arrival time: denotes the shortest inter-arrival time observed.

Average inter-arrival time: calculates the average inter-arrival time among items with the lowest inter-arrival time.

We classify the above into two categories based on their relation to recency or frequency. The set of recency indicators comprises the minimum and average inter-arrival times, whereas the set of frequency indicators includes the rest (unique count, number of most accessed items, maximum frequency, average frequency, and frequency distribution).

Sketches Component. We utilize several sketch data structures, namely Space Saving (SS) [38], Count-Min Sketch [16], HyperLogLog [22], and our novel ISketch. Each sketch is configured with identical values of ϵ and θ, resulting in a total space overhead of $O(\epsilon^{-1} \log \delta^{-1})$, as depicted in Table 2 (all constants in the table are small). We maintain all sketches upon arrival and feed all of their information into the learning model. Consequently, the estimation error associated with each indicator is ϵW, where W denotes the frame size. Our experiments, as discussed in Section 4, demonstrate that this estimation error has minimal impact on prediction accuracy.

Deriving Statistical Indicators from Sketches. For SS, item frequency is determined by maintaining a collection of counters, as described in Section 2.2. The *number of most accessed items* indicator is obtained by counting the items in the collection with a count exceeding θW, where W represents the frame size and $\theta \in [0, 1]$ is a frequency threshold. Further, SS maintains two variables: the maximum and average counter values, which provide the *maximum frequency* and *average frequency* indicators, respectively. Using the HyperLogLog algorithm from [22], we can estimate the *unique count*. At the end of each frame, we generate the frequency vector from the CMS sketch array to obtain the *frequency distribution* indicator. Each entry i represents the count of frequency i occurrences. Section 3.2 below elaborates on the retrieval of the *minimum* and *average inter-arrival times* indicators using our ISketch sketch.

3.2 ISketch: Interarrival Sketch

To our knowledge, we are the first to use sketches to approximate recency in a workflow. To this end, we present the Interarrival Sketch (ISketch) algorithm, specifically designed for tracking inter-arrival times. The inter-arrival time of an item x refers to the time elapsed between its last two occurrences within the system. This measurement has been widely recognized as one of the preferred methods for quantifying workload recency [27, 40]. However, maintaining precise inter-arrival times for each item consumes a significant amount of storage. Therefore, ISketch focuses on tracking inter-arrival times only for items with low inter-arrival values, as these items have the greatest impact on estimating workload recency. To accomplish this without knowing these items' identities, ISketch employs a Space Saving-inspired table structure with k entries. Each entry consists of an item ID, inter-arrival time, and last arrival timestamp.

Upon the arrival of an item x, if x has no allocated entry, we replace the item whose inter-arrival value is maximal in the table with x and reset its associated fields. Alternatively, if x already has an allocated entry, ISketch calculates its last inter-arrival time by subtracting the last arrival timestamp from the current timestamp. If the result is lower than the inter-arrival field, we update the inter-arrival value. The QuerySTAT() function retrieves the average and minimum inter-arrival times of all items in the trace whose inter-arrival times are shortest. We maintain the average inter-arrival time and minimum value with each arrival and satisfy QuerySTAT() by returning these pre-computed values.

ISketch includes the exact inter-arrival times of items that exhibit frequent arrivals and possess short recency periods. Specifically, when an item arrives at least twice and its inter-arrival time does not exceed the maximum time stored in ISketch, it is allocated an entry within the data structure. Once an item is inserted, it remains inside ISketch as long as its inter-arrival time remains less than the maximum value. This condition holds for items that consistently have a short inter-arrival time. On the other hand, ISketch replaces items with infrequent arrivals, resulting in their inter-arrival time no longer being tracked. Consequently, ISketch does not provide an error bound on inter-arrival times for unmonitored items. Moreover, low-frequency items encountered towards the end of the workload have a higher probability of being monitored by ISketch. This is because items that were frequently accessed in the past but have not been accessed for an extended period are likely to be evicted from ISketch to make room for items with shorter recent inter-arrival times. For a detailed understanding of the ISketch algorithm, refer to Algorithm 1 in the appendix.

▶ **Lemma 1.** *An item with an inter-arrival time* $< interArr_{max}$*, must exist in the SS table.*

▶ **Theorem 2.** *IS requires* $k(2\log W + \log\frac{W}{k} + 2\omega)(1 + o(1))$ *space where* k *the number of entries,* ω *is the number of bits required to represent an item in* $\mathcal{R}$*, and* W *is the frame size. ISketch performs updates and answers queries as well as computes* QUERYSTAT() *in* $O(1)$ *time.*

The proof of Theorem 3.2 is in the appendix.

3.3 ML submodel

We use a neural network (NN) as the ML submodel. In particular, we use a 4-layer fully-connected NN [24] with two hidden layers and ReLU activation that implements multi-class regression. Formally, denote the output of a 4-layer fully connected neural network as: $N_{i,j}(x) = A(A(x \cdot w_1 + b_1) \times w_2 + b_2) \times w_3 + b_3$, where x is the input (statistical indicators), w_1, b_1 are the weight and bias vectors for layer 1 (first hidden layer), w_2, b_2 are the weight and bias vectors for layer 2 (second hidden layer), and w_3, b_3 are the weight and bias vectors for the output layer. The ReLU function A applies a function a on each element of an input vector: $a(x) = \left\{ \begin{array}{ll} x, & x \geq 0 \\ 0, & x < 0 \end{array} \right\}$. The submodel output, denoted $M_i, j(x)$, is defined as: $M_{i,j}(x) = H(N_{i,j}(x))$, where H is the softmax function, which generates a probability distribution for our policy classes given their respective prediction scores. The result of H is a vector with elements in $[0, 1]$ that all sum up to 1.

The training process of ALPS is depicted in Figure 2. To train the NN submodel ($model_{ml}$), we utilize traces from real-world workloads and employ supervised learning with a cross-entropy loss function. For feature generation and label creation, we partition each trace into frames and retain the four sketches, namely SS_{train}, HLL_{train}, CM_{train}, and $ISketch_{train}$. We also utilize a caching simulator [36] to simulate various heuristic caching algorithms. At the end of a frame, we take the statistical indicators from the sketches and use them as features. Ground-truth hit ratios for each caching algorithm are obtained from the simulator. Users of ALPS can either directly use our pre-trained model, utilize our model as a starting point for further training, or perform training from scratch using their own traces.

3.4 Frame Size Selection

Determining the optimal frame size is a critical component of the current challenge. In order to perform the task, the user must have a preliminary understanding of the temporal and scalar alterations within the data stream. Several research works have delved into the domain of detecting alterations within data streams, as discussed in [1, 32, 49]. These methodologies fundamentally depend on an understanding of the inherent probability distribution, which forms the basis of the incoming data stream. However, due to the inherently unpredictable dynamics of data streams, acquiring preliminary information is rarely straightforward.

Intuitively, when the frame size is small, the sketch counters are relatively small, lacking statistical significance. In such a scenario, the sketch counters are so close together that it becomes difficult to distinguish real differences between them. Such a situation results in an inaccurate approximation of the relevant statistical indicators. Conversely, exceedingly large frames may hide dynamic transformations occurring within a singular frame. In general, if a user assigns a frame size that either exceeds or falls short of the necessary scale, the frame fails to effectively reflect the latest workload changes. The notion of frame size can be discussed in two contexts: as a hyperparameter or as a model parameter. According to the first perspective, the frame size is adjusted to a value proportionate to the cache size. Alternatively, in the second perspective, the frame size is viewed as a parameter, and its

determination is subject to the model's learning capabilities. For our experimental setup, the default frame size is set to ten times the cache size; we analyze the effect of the frame size on the hit ratio in Section 4.

3.5 Putting All Together

We summarize all the steps of ALPS:

Training: (1) Divide the trace into frames and maintain the sketches for each frame. (2) Run the cache simulator with the various caching algorithms. (3) Train on each frame.

Cache selection (inference): (1) Query the NN submodel at the end of the frame. (2) Configure the cache policy according to the returned policy when it differs from the current cache policy.

ALPS pseudocode appears in Algorithm 2 in the appendix.

4 Evaluation

In this section, we compare ALPS with five heuristic caching algorithms: LFU, LRU, Hyperbolic, ARC, and W-TinyLFU, and with the state-of-the-art learning algorithms LRB [47] and CACHEUS [43].

4.1 Implementation

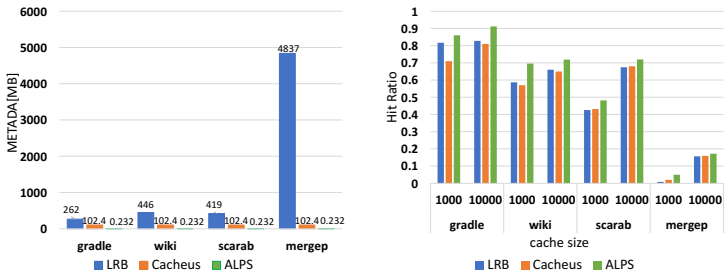
We have implemented ALPS in C++ and employed Caffeine's simulator [36] to derive cache hit ratios for various management policies. Our simulations encompassed five cache management algorithms, namely LFU [44], LRU [25], Hyperbolic [9], ARC [37], and W-TinyLFU [20]. The implementations of these algorithms were sourced from the Caffeine project's repository.

Regarding the SS, HLL, and CM sketches, we integrated the original author's implementation and realized ISketch's implementation ourselves. As previously mentioned, we utilized a fully connected MLP neural network (NN) consisting of two hidden layers with ReLU activation. The input features were constructed using statistical indicators introduced in Section 3.1, combined with a frequency distribution vector of length 32. The hidden layers comprised 512 neurons each, and the output size corresponded to the number of policies. The output layer consisted of 64 neurons. The total number of trainable parameters in this NN was computed as $38 \times 512 + 512 \times 512 + 512 \times 64 + 512 + 512 + 64 = 315,456$.

For training the NN submodel, we employed PyTorch [41] and trained it on various real-world workloads using supervised learning with cross-entropy error loss function. The labels for the training phase were obtained from the Caffeine simulator [36]. To be more specific, we partitioned each workload into frames based on the customizable frame size option. Subsequently, we executed the Caffeine simulator to calculate the per-frame hit ratio for each policy, identify the optimal policy within each frame, and assign the corresponding label to the frame.

4.2 Experimental Setup

The experiments were run on a DGX-A100 cluster using Slurm to schedule work. Each job was limited to a single A100 card with 40GB memory. It is important to note that ALPS can also be run on CPUs due to its low computational and memory overheads.



(a) Metada overhead. **(b)** Hit Ratio.

Figure 3 (a) LRB, CACHEUS and ALPS memory overhead using various traces (b) LRB, CACHEUS and ALPS hit ratio using previously stated traces separately with two cache sizes $(1000, 10000)$. ALPS's frame size is 10000 items.

Traces. Our evaluation is based on real-world workloads from a variety of different domains: databases, analytic systems, transaction processing, search engines, and Windows servers are just a few examples. These workloads exhibit a wide range of underlying characteristics: some display a strong bias towards recency, others exhibit a bias towards frequency, and some present a combination of both. It is important to note that we meticulously partitioned the traces into separate subsets for training, validation, and testing purposes. To prevent overfitting, we trained our model on the interleaved prefixes of all traces. Subsequently, we conducted independent testing on each suffix of every trace. The workloads reported below include:

P1–P14, Mergep: 14 traces obtained from Windows NT workstations using Vtrace, which captures disk operations with the use of device filters. The traces were collected over a period of several months [37], containing about 491 million accesses.

Gradle: A trace from a distributed build cache, the Gradle project, that holds the compiled output so that subsequent builds on different machines can fetch the results instead of building anew. Since machines leverage local build caches, the distributed cache is recency-biased as only the latest changes are requested. It includes $\approx$ 2M accesses.

Scarab: A one-hour trace from Scarab Research of product recommendation lookups for several e-commerce sites of varying sizes worldwide. This trace includes $\approx$ 28M accesses.

Wikipedia: Wikipedia trace containing 10% of the traffic to Wikipedia during September–October 2007 [50]. This trace size is about 12 million accesses.

Cache configurations. Cache hit rate is the prime metric when evaluating caching algorithms. We compare our algorithm's performance across a range of cache and frame sizes when handling the above traces.

Sketches overhead. The memory overhead for ALPS is dominated by four sketches. We configure each sketch with the same ϵ and δ (if needed) values. We set the number of entries (k) for ISketch to ϵ^{-1}. As shown in Table 2, the overall space overhead of all sketches is $O(\epsilon^{-1} \log \delta^{-1})$.

Comparison to LRB and CACHEUS. We compare ALPS to LRB and CACHEUS, considering different cache sizes of 1000 and 10000. We examined the hit ratio and memory overhead of these algorithms. The implementation code for LRB was obtained from [46], while the code for CACHEUS was sourced from [48].

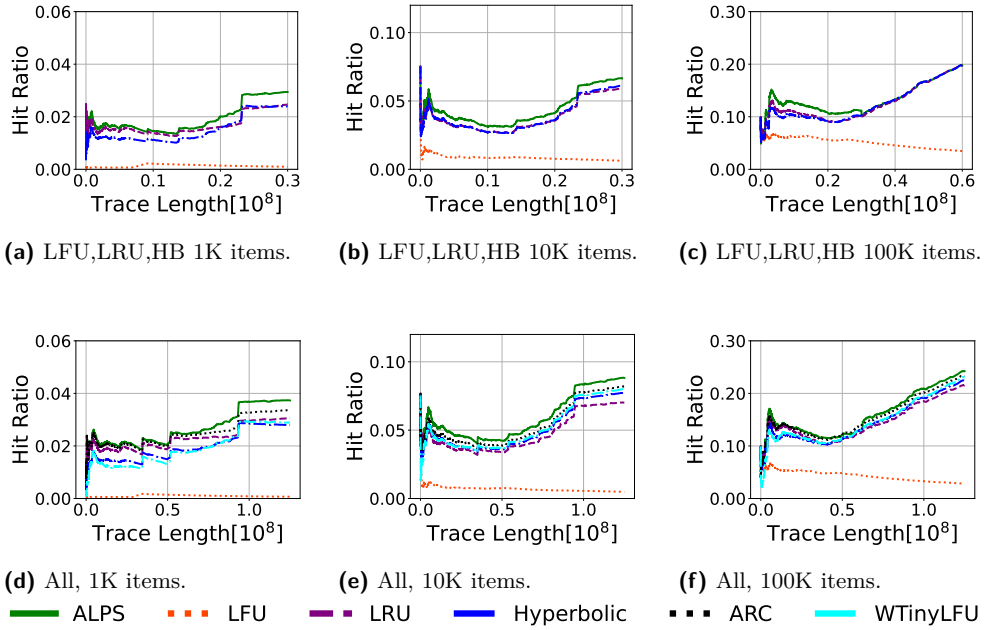
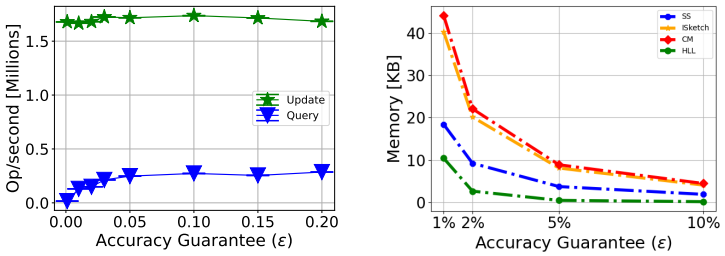


Figure 4 Detailed comparison of ALPS against LFU, LRU, Hyperbolic, ARC and W-TinyLFU in 3 cache sizes when each frame contains $10 * \textit{cache size}$ accesses when ALPS is trained on (LFU, LRU, Hyperbolic), and on all the five policies (LFU, LRU, Hyperbolic, ARC, W-TinyLFU). The plots show the hit ratio as a function of the trace length. The used trace in this experiment is **mergep**, which is a merge of 14 traces obtained from Windows NT workstations. The first column has a cache size of 1K items, the second column has a cache size of 10K items, and the third column has a cache size of 100K items.

Figure 3a illustrates the memory overhead of the LRB, CACHEUS, and ALPS policies across four distinct traces. In this experiment, the cache size was set to 100000 items, and the frame size for ALPS was also 100000 items. It is evident that ALPS demonstrated superior memory efficiency than the other algorithms. LRB maintains item-specific information within a sliding memory window, resulting in memory overhead that varies depending on the cache size and average item size in a trace. On the other hand, CACHEUS consumes approximately twice the cache size in memory for metadata, which tracks cache-resident items and historical items. Therefore, CACHEUS memory overhead primarily depends on cache size. In contrast, ALPS utilizes frame-specific information derived from compact sketches, making it insensitive to cache size and average item size in a trace. Consequently, ALPS's memory overhead remains constant across all traces, while LRB's memory overhead varies significantly, reaching magnitudes higher than ALPS.

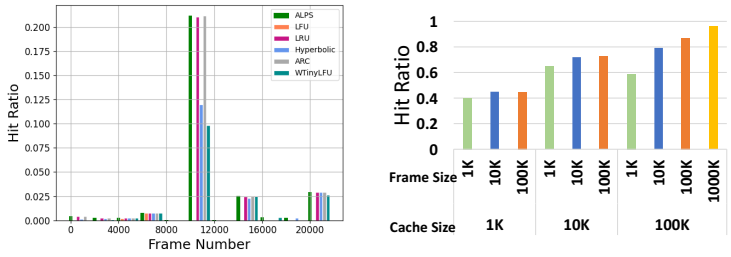
Additionally, Figure 3b presents a comparison of the hit ratios achieved by LRB, CACHEUS, and ALPS using cache sizes of 1000 and 10000 using previously stated traces separately. In this comparison, ALPS was trained using the five heuristic-based algorithms, and its frame size set to 10000 items. It is notable that ALPS consistently outperformed both LRB and CACHEUS in terms of hit ratio, particularly when the cache size was small.

Hit ratio. We conducted experiments to measure the hit ratio as a function of trace length using the Mergep trace (P1-P14). The page size for these traces was set to 512 bytes, while the frame size was defined as 10 times the cache size. The hit ratios for different cache sizes



(a) Sketches performance. **(b)** Sketches overhead.

Figure 5 (a) Sketches update and query performance of the four sketches (SS, ISketch, CM and HLL) using the previously stated traces and with frame size of 10K items.(b) Sketches space comparison as a function of ϵ when the frame size to 10000 items.



(a) Hits per frame. **(b)** Hits vs. frame sizes.

Figure 6 (a) Hit ratio of ALPS, LFU, LRU, Hyperbolic, ARC and W-TinyLFU per sampled frames when ALPS is trained on the five policies with cache of size 100K using mergep trace (b) Hit ratio of ALPS using scarab trace with different cache sizes (1K,10K,100K items) and different frame sizes.

are presented in Figure 4, where ALPS was trained on (LFU, LRU, Hyperbolic) as well as on all five policies (LFU, LRU, Hyperbolic, ARC, W-TinyLFU). The first column represents a cache size of 1K items, the second column corresponds to a cache size of 10K items, while the third represents a cache size of 100K items.

Our findings indicate that ALPS achieves the highest hit ratio when trained on all five policies with a cache size of 100K items. LFU has the lowest hit ratio. As expected, the hit rate for all algorithms increases with cache size. Notably, Figure 4 demonstrates that training ALPS on all five policies yields a more substantial improvement in the hit ratio than training it solely on (LFU, LRU, Hyperbolic). This is because ALPS benefits from a wider range of caching algorithms adapted to the frames' features. This observation is evident when comparing Figure 4a with Figure 4d, for example. Furthermore, the improvements in the hit ratio are more pronounced for small caches, as illustrated by the comparison of the columns in Figure 4. These enhancements arise from smaller caches, resulting in smaller frames and more frequent evictions. To that end, ALPS dynamically adjusts the cache management strategy, benefiting from the varying choices of evicted items by the heuristic-based policies.

In Table 3, we present the percentage increase in the hit ratio achieved by ALPS over LRU, Hyperbolic, ARC, and W-TinyLFU. The evaluations were performed using the aforementioned traces and a frame size of 10 times the cache size for three cache sizes: 1K items, 10K items, and 100K items. The results demonstrate that ALPS outperforms the other algorithms across all cache sizes. The most significant improvements were observed in the smallest cache size of 1K items. As mentioned earlier, ALPS enhances the hit ratio by predicting

Table 3 Hit ratio increase by ALPS in three cache sizes (1K, 10K, and 100K items) against LRU, Hyperbolic, ARC and WTinyLFU when trained on (LFU, LRU,Hyperbolic), (ARC, W-TinyLFU), and with all the five policies using the previously stated traces with frame size of $10 * |cache|$ items.

Training Set	Cache Size	LRU	Hyperbolic	ARC	WTinyLFU
LFU, LRU, HB	1000	18.9%	22.4%	–	–
LFU, LRU, HB	10000	10.6%	6.7%	–	–
LFU, LRU, HB	100000	5%	1%	–	–
ARC, WTinyLFU	1000	–	–	10.4%	18.8%
ARC, WTinyLFU	10000	–	–	3.4%	4%
ARC, WTinyLFU	100000	–	–	2.6%	5%
All	1000	20%	31%	9%	26%
All	10000	19.3%	8.4%	2.17%	4.84%
All	100000	19.1%	14.5%	9.8%	11.5%

the best-performing cache policy for the next frame and dynamically configuring cache management accordingly. This behavior is particularly impactful in smaller caches due to smaller frames and increased evictions. This results in larger differences in heuristic-based policy choices. Specifically, the hit ratio percentage increase of ALPS compared to LRU is 20%, Hyperbolic is 31%, ARC is 9%, and W-TinyLFU is 26%.

Figure 6a depicts the hit ratios on the Mergep trace for ALPS, LFU, LRU, Hyperbolic, ARC, and W-TinyLFU when ALPS is trained on the five policies, using a cache size of 100K items, as a function of frame number. To enhance clarity, only samples of frames are displayed. The hit ratio obtained by ALPS in the sampled frames closely aligns with the highest hit ratio.

Effect of frame size. Figure 6b illustrates the impact of frame size on the hit ratio of ALPS with varying cache sizes (1K, 10K, 100K items) using the Scarab trace. As expected, both cache and frame sizes positively influence the hit ratio. Comparing the frame sizes with a 100K cache, we observe that smaller frames (relative to the cache size) lead to a reduced hit ratio. This is due to statistical indicators exhibiting minimal variations between two small frames. Conversely, extremely large frames may compromise the efficiency of the "winning" heuristic caching algorithm at the beginning of the frame. This indicates that it is advisable to avoid excessively long frames, as demonstrated in the case of 1K cache items.

Inference time of the NN submodel. Regarding the inference time of the NN submodel in ALPS, it is worthwhile to note that inference is executed only once per frame. The inference time represents the forward propagation duration. To ensure synchronized execution, we implement synchronization between the host (CPU) and the device (GPU) to record time only after GPU-based activity. This is achieved by performing a "GPU warm-up" by running dummy examples, which initializes the GPU and prevents it from entering power-saving mode during time measurement. As a result, we employ tr.cuda.event to measure GPU time. Table 4 in the appendix presents the inference time of the model for various training policies: (LFU, LRU), (LFU, LRU, Hyperbolic), (ARC, W-TinyLFU), and all five policies (LFU, LRU, Hyperbolic, ARC, W-TinyLFU). The values represent the mean of 300 iterations used to compute the inference time. It is observed that the inference time increases as the number of training policies expands. For instance, when ALPS is trained on five policies with a frame size of 10^4, the inference is performed once every 10^4 access and takes 0.309 milliseconds.

Note that while these measurements were obtained using a GPU, the inference in ALPS can also be executed on CPUs, due to its moderate computational and memory overheads.

Sketches overhead. Figure 5b presents the space occupied by the sketches for a given ϵ, using the previously mentioned traces. The number of entries (k) for ISketch was set to ϵ^{-1}. It is evident that as ϵ decreases, all sketches require more space. In comparison to the SS sketch, ALPS consumes additional space because ISketch maintains the exact item IDs instead of their fingerprints. This is necessary to accurately report inter-arrival times for items with low inter-arrival times.

The performance overhead of the update and query operations for the four sketches (SS, ISketch, CM, and HLL) is depicted in Figure 5a. The update performance is exceptionally efficient and remains unaffected by changes in ϵ values, as the update complexity is reported to be $O(1)$, as shown in Table 2. However, query performance in certain sketches depends on ϵ. As ϵ decreases, more entries are included in the SS sketch, leading to slower query performance. Yet a query is executed only once per frame.

5 Conclusions

We have introduced ALPS, a lightweight cache management meta-policy that effectively combines ML with traditional heuristic-based approaches while addressing memory overhead challenges associated with ML components. The key idea behind ALPS is to divide the access sequence into frames and employ ML to predict the most effective cache management policy for each frame. Unlike existing ML caching algorithms that store item-specific information within a sliding window, ALPS utilizes space-efficient sketches to maintain per-frame information, making it a highly resource efficient algorithm. The features derived from these sketches are fed into the ML process, which is invoked only once per frame.

We have also introduced ISketch, an efficient algorithm for tracking inter-arrival times, enabling us to efficiently monitor frame recency. Subsequently, we presented the design, implementation, and evaluation of ALPS. Our experiments clearly demonstrate that ALPS significantly improves the hit ratio across various cache sizes when compared to traditional heuristic-based approaches, as well as state-of-the-art learning algorithms such as LRB and CACHEUS. Furthermore, the memory overhead of ALPS is orders of magnitude smaller than that of LRB and CACHEUS. As part of our future work, we plan to expand ALPS training to incorporate additional heuristic-based caching algorithms. This will enable the ML process to predict the optimal cache size and support weighted items. This will enhance ALPS's capabilities in optimizing cache performance. All code is available online [45].

References

1 Charu C Aggarwal. A framework for Diagnosing Changes in Evolving Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 575–586, 2003. doi:10.1145/872757.872826.

2 Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating Content Management Techniques for Web Proxy Caches. In *In Proc. of the 2nd Workshop on Internet Server Performance*, 1999.

3 Martin Arlitt, Rich Friedrich, and Tai Jin. Performance Evaluation of Web Proxy Cache Replacement Policies. *Perform. Eval.*, 39(1-4):149–164, feb 2000. doi:10.1016/S0166-5316(99)00062-0.

4 Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *Proc. of the 3rd USENIX Conf. on File and Storage Technologies (FAST)*, pages 187–200, 2004. URL: http://www.usenix.org/events/fast04/tech/bansal.html.

5 Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 389–403, 2018. URL: https://www.usenix.org/conference/nsdi18/presentation/beckmann.

6 Nathan Beckmann and Daniel Sanchez. Talus: A Simple Way to Remove Cliffs in Cache Performance. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75, 2015. doi:10.1109/HPCA.2015.7056022.

7 L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966. doi:10.1147/SJ.52.0078.

8 Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy Hitters in Streams and Sliding Windows. In *The 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–9, 2016. doi:10.1109/INFOCOM.2016.7524364.

9 Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 499–511, 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein.

10 Sara Bouchenak, Alan Cox, Steven Dropsho, Sumit Mittal, and Willy Zwaenepoel. Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2006. doi:10.1007/11925071_1.

11 Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of the 18th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM)*, pages 126–134, 1999. doi:10.1109/INFCOM.1999.749260.

12 Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. EnclaveCache: A Secure and Scalable Key-Value Cache in Multi-Tenant Clouds Using Intel SGX. In *Proc. of the 20th ACM/IFIP International Middleware Conference*, pages 14–27, 2019. doi:10.1145/3361525.3361533.

13 Ludmila Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Technical report, In HP Tech. Report, 1998.

14 Gregory V. Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2000. doi:10.1007/3-540-45559-0_1.

15 Wonil Choi, Bhuvan Urgaonkar, Mahmut Taylan Kandemir, and George Kesidis. Multi-Resource Fair Allocation for Consolidated Flash-Based Caching Systems. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, pages 202–215, 2022. doi:10.1145/3528535.3565245.

16 Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *Journal of Algorithms*, 55(1):58–75, apr 2005. doi:10.1016/J.JALGOR.2003.12.001.

17 Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 24–44, 2000. doi:10.1007/3-540-45559-0_2.

18 Dgraph. Ristretto: A High Performance Memory-Bound Go Cache, 2020. URL: https://github.com/dgraph-io/ristretto.

19 Xiaoming Du and Cong Li. SHARC: Improving Adaptive Replacement Cache with Shadow Recency Cache Management. In *Proc. of the 22nd ACM/IFIP International Middleware Conference*, pages 119–131, 2021. doi:10.1145/3464298.3493389.

20 G. Einziger, R. Friedman, and B. Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage (TOS)*, 2017. doi:10.1145/3149371.

21 Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive Software Cache Management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018. doi:10.1145/3274808.3274816.

22 Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156, 2007.

23 Priya Gupta, Nickolai Zeldovich, and Samuel Madden. A Trigger-Based Middleware Cache for ORMs. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2011. doi:10.1007/978-3-642-25821-3_17.

24 Simon Haykin. *Neural Networks: a Comprehensive Foundation*. Prentice Hall PTR, 1994.

25 John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

26 Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an Effective Improvement of the CLOCK Replacement. In *Proc. of the USENIX Annual Technical Conference (ATC)*, 2005. URL: http://www.usenix.org/events/usenix05/tech/general/jiang.html.

27 Song Jiang and Xiaodong Zhang. LIRS: an Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proc. of the International Conference on Measurements and Modeling of Computer Systems SIGMETRICS*, pages 31–42, jun 2002. doi:10.1145/511334.511340.

28 Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of the 20th Int. Conf. on Very Large Data Bases (VLDB)*, pages 439–450, 1994. URL: http://www.vldb.org/conf/1994/P439.PDF.

29 G. Karakostas and D. N. Serpanos. Exploitation of Different Types of Locality for Web Caches. In *Proc. of the 7th Int. Symposium on Computers and Communications (ISCC)*, pages 207–212. IEEE, 2002. doi:10.1109/ISCC.2002.1021680.

30 Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching Strategies to Improve Disk System Performance. *Computer*, 27(3):38–46, 1994. doi:10.1109/2.268884.

31 Tatsuya Kawano. A High Performance Concurrent Caching Library for Rust, 2021. URL: https://github.com/moka-rs/moka.

32 Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting Change in Data Streams. In *VLDB*, volume 4, pages 180–191, 2004. doi:10.1016/B978-012088469-8.50019-X.

33 Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Computers*, 50(12):1352–1361, 2001. doi:10.1109/TC.2001.970573.

34 Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and Analysis of a Container-Based Flash Cache for Compound Objects. *ACM Trans. on Storage (ToN)*, 13(3), sep 2017. A preliminary version appeared in ACM/IFIP Middleware 2015. doi:10.1145/3094785.

35 Tanu Malik, Xiaodan Wang, Philip Little, Amitabh Chaudhary, and Ani Thakar. A Dynamic Data Middleware Cache for Rapidly-Growing Scientific Repositories. In *Proc. of the ACM/IFIP/USENIX 11th International Conference on Middleware*, pages 64–84, 2010. doi:10.1007/978-3-642-16955-7_4.

36 Ben Manes. Caffeine: A High Performance Caching Library for Java 8. *https://github.com/ben-manes/caffeine*, 2017.

37 Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of the 2nd USENIX Conf. on File and Storage Technologies (FAST)*, pages 115–130, 2003. URL: http://www.usenix.org/events/fast03/tech/megiddo.html.

38 Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient Computation of Frequent and Top-K Elements in Data Streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005. doi:10.1007/978-3-540-30570-5_27.

39 Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *ACM SIGMOD Rec.*, 22(2):297–306, jun 1993. doi:10.1145/170035.170081.

40 Sejin Park and Chanik Park. FRD: A Filtering Based Buffer Cache Algorithm that Considers both Frequency and Reuse Distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

41 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

42 Redis-Labs. Using Redis as an LRU cache, 2020. URL: `https://redis.io/topics/lru-cache`.

43 Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST)*, pages 341–354, 2021. URL: `https://www.usenix.org/conference/fast21/presentation/rodriguez`.

44 Dimitrios N Serpanos, George Karakostas, and Wayne Hendrix Wolf. Effective Caching of Web Objects Using Zipf's Law. In *IEEE International Conference on Multimedia and Expo (ICME): Latest Advances in the Fast Changing World of Multimedia (Cat. No. 00TH8532)*, volume 2, pages 727–730, 2000. `doi:10.1109/ICME.2000.871464`.

45 Rana Shahout. Open Source Code. URL: `https://anonymous.4open.science/r/sketch_learning-7A60`.

46 Zhenyu Song. webcachesim2: A Simulator for CDN Caching and Web Caching Policies, 2019. URL: `https://github.com/sunnyszy/lrb`.

47 Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 529–544, 2020. URL: `https://www.usenix.org/conference/nsdi20/presentation/song`.

48 Systems Research Laboratory (SyLab). Cacheus Project, 2021. URL: `https://github.com/sylab/cacheus`.

49 Yingying Tao and M Tamer Ozsu. Mining Data Streams with Periodically Changing Distributions. In *Proceedings of the 18th ACM conference on Information and Knowledge Management*, pages 887–896, 2009. `doi:10.1145/1645953.1646065`.

50 Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Computer Networks*, 53(11):1830–1845, 2009. `doi:10.1016/J.COMNET.2009.02.019`.

51 Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *USENIX Annual Technical Conference (ATC)*, pages 487–498, 2017. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger`.

A ISketch and ALPS Algorithms

A.1 ISketch Pseudocode

Algorithm 1 ISketch.

```
    Init: ts ← 0, interArr_min ← 0, interArr_max ← 0, avrg ← 0
 1: function INSERT(x, ℓ)
 2:     ts ← ts + 1
 3:     if x is monitored then
 4:         x.interArr ← ts − x.lastArr
 5:         x.lastArr ← ts
 6:         update interArr_min, avrg
 7:     else
 8:         if Less than k items are monitored then
 9:             x.lastArr ← ts
10:             x.interArr ← ∞
11:             update interArr_min, avrg
12:         else
13:             Let x′ be the element with largest inter-arrival
14:             Start monitoring x instead of x′;
15:             x′.lastArr ← ts
16:             x′.interArr ← x.interArr
17:             update interArr_min, avrg
18:         end if
19:     end if
20: end function

21: function QUERYSTAT()
22:     return (interArr_min, avrg)
23: end function

24: function QUERY(x)
25:     if x is monitored then
26:         return x.interArr
27:     else
28:         return interArr_max
29:     end if
30: end function
```

A.2 Proof of Theorem 3.2

The ISketch implementation is built on the CSS implementation [8] with k entries. It maintains k entries with three values: item ID counters, latest arrival timestamps, and inter-arrival times. In this implementation, we substitute the frequency counter with inter-arrival time and add the last arrival timestamp to each allocated entry in the ID-Index data structure, which adds $k \log W$ to the space overhead compared to CSS. We have $k(2 \log W + \log \frac{W}{k} + 2\omega)(1 + o(1))$ space in total. The implementation in [8] allows item additions and point queries in $O(1)$ time (w.h.p.). The ISketch implementation is symmetric to CSS, with the exception that we replace the maximal counter rather than the minimal counter and keep the average and minimal inter-arrival time updated on each arrival, which also takes $O(1)$ time. QUERYSTAT() simply returns the average and minimum inter arrival times that have been kept. In total, the three operations take $O(1)$ time.

A.3 ALPS Pseudocode and Inference Time

Algorithm 2 ALPS.

Initialization: $fs \leftarrow 0$, *initialize* $model_{ml}, SS, HLL, ISketch, CM,$
$SS_{train}, HLL_{train}, ISketch_{train}, CM_{train}$.
function TRAIN()
  for $trace \in traces$ **do**
    break trace into frames
    for $req \in frame$ **do**
      $UpdateSketches(SS_{train}, HLL_{train}, ISketch_{train}, CM_{train})$
    end for
    Run caching simulator
    if frame ends **then**
      $features$ = *Extract from* $SS_{train}, HLL_{train}, ISketch_{train}, CM_{train}$
      $labels$ = *Get labels from the caching simulator*
      train $model_{ml}$ *with* $(features, labels)$
    end if
  end for
end function

function UPDATE(Req_i)
  $fs \leftarrow (fs + 1) \mod W$
  $SS.Add(Req_i)$
  $HLL.Add(Req_i)$
  $ISketch.Add(Req_i)$
  $CM.Add(Req_i)$
  if $fs \mod W = 0$ **then**
    $ConfigureCache()$
    flush sketches
  end if
end function

function CONFIGURECACHE()
  $features = GetFeatures(SS, HLL, ISketch, CM)$
  $policy_{new} = model_{ml}(features)$
  if $Cache.GetPolicy() != policy_{new}$ **then**
    $Cache.SetPolicy(policy_{new})$
  end if
end function

Table 4 Mean inference time of ALPS when trained on (LFU, LRU), (LFU, LRU, Hyperbolic), (ARC, W-TinyLFU), and finally with all five policies (LFU, LRU, Hyperbolic, ARC, W-TinyLFU) with frame size of 10000 items. Inference is executed only once per frame.

Trained Policies	Mean Inference Time
LFU, LRU	0.304 milliseconds
LFU, LRU, Hyperbolic	0.308 milliseconds
ARC, WTinyLFU	0.3026 milliseconds
All	0.309 milliseconds

Atomic Register Abstractions for Byzantine-Prone Distributed Systems

Vincent Kowalski
LS2N, Nantes Université, France

Achour Mostéfaoui
LS2N, Nantes Université, France

Matthieu Perrin
LS2N, Nantes Université, France

Abstract

The construction of the atomic register abstraction over crash-prone asynchronous message-passing systems has been extensively studied since the founding work of Attiya, Bar-Noy, and Dolev. It has been shown that $t < n/2$ (where t is the maximal number of processes that may be faulty) is a necessary and sufficient requirement to build an atomic register. However, little attention has been paid to systems where faulty processes may exhibit a Byzantine behavior. This paper studies three definitions of linearizable single-writer multi-reader registers encountered in the state of the art: Read/Write registers whose `read` operations return the last written value, Read/Write-Increment registers whose `read` operations return both the last written value and the number of previously written values, and Read/Append registers whose `read` operations return the sequence of all previously written values. More specifically, it compares their computing power and the necessary and sufficient conditions on the maximum ratio t/n which makes it possible to build reductions from one register to another. Namely, we prove that $t < n/3$ is necessary and sufficient to implement a Read/Write-Increment register from Read/Write registers whereas this bound is only $t < n/2$ for a reduction from a Read/Append register to Read/Write-Increment registers. Reduction algorithms meeting these bounds are also provided.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Process synchronization; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Byzantine processes, Concurrent Object, Linearizability, Shared Register

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.35

Related Version *Full Version*: `https://hal.science/hal-04213718/`

Funding ByBloS ANR project (ANR-20-CE25-0002): "BeYond BLockchainS, Modular Building Blocks for Large-Scale Trustless Multi-users Apps".
PricLeSS Project, funded by the COMINLabs Labex Project (ANR-10-LABX-0007): "Privacy-Conscious Legally-Sound blockchain Storage".

1 Introduction

Atomic register abstractions. The register abstraction is the basis of the Turing machine's tape. It provides two basic operations: a write operation that allows defining a new value for the register and a read operation that returns its value. In concurrent architectures such as multi-core systems, the read/write semantics of a register is the cleanest and most easy-to-understand abstraction of shared memory and is extensively used in multi-threaded programs. In such a setting, a register that can be accessed concurrently by several processes represents a communication medium. In a message-passing system where processes may

experience crash failures, Attiya, Bar-Noy, and Dolev [5] proposed the first emulation of a shared register (called ABD) for which it has been shown that $t < n/2$ (where n is the total number of processes and t is the maximal number of processes that may crash) is a necessary and sufficient requirement. Several algorithms have been proposed in order to enhance space or time efficiency.

According to which process is allowed to read or write a register and the significance of the returned value, several types of registers have been proposed, among which single-writer multi-reader (SWMR) and multi-writer multi-reader (MWMR) registers. Several levels of consistency can also be proposed: atomic, regular, and safe. A register is said to be atomic (or linearizable) if (a) each read or write operation appears as if it has been executed instantaneously at a single point of the timeline, between its start event and its end event, (b) no two operations appear at the same point of the timeline, and (c) a read returns the value written by the closest preceding write operation (or the initial value of the register if there is no preceding write) [17]. Reduction algorithms from one type of register to another (MWMR vs SWMR and atomic vs regular or safe) have been proposed (these registers are thus equivalent from a computability point of view).

Byzantine-prone distributed systems. The implementation of shared registers has been first studied in the crash failure model and then extended to Byzantine failures. A Byzantine process is a process that may deviate from its specification [22]. Byzantine faults gained interest since the work on Byzantine Fault Tolerance (BFT) [9], an implementation of a replicated state machine over a set of servers [23]. More recently, Blockchains made their appearance and a link was quickly made with the BFT approach [2]. Blockchains can be seen as eventually consistent implementations of a ledger data structure that consider in addition the semantic chaining between the different blocks and cryptography is used as a tool. The ledger itself can be seen as a particular register. In this paper, we propose to study register abstractions, the basic data structure, in distributed systems prone to Byzantine faults. Several works have addressed the design of a distributed shared storage in the client/server model prone to Byzantine failures [1, 10, 20]. A set of server processes implements a shared storage abstraction accessed by client processes. The different processes are, thus, separated into two classes and the system is not symmetric. While some servers can be Byzantine, most papers restrict the type of failure allowed to clients. [1] considers clients that can only crash, and [6] considers that clients can be Byzantine but a bounded number of times. On the other hand, [19] considered the use of signatures, and [13, 14] explored the conditions under which one can have fast reads (one-way messages) when servers never communicate with each other. Finally, [3] considers that readers can be Byzantine but not the writer.

In the context of asynchronous message-passing systems where at most t processes out of the n processes of the system can exhibit a Byzantine behavior, only the implementation of atomic SWMR (single-writer multi-reader) registers has been considered due to the fact that a Byzantine process can corrupt any register it can write. As Cohen and Keidar phrase it "In practice, multi-writer multi-reader (MWMR) registers are useless in a Byzantine environment as an adversary that controls the scheduler can prevent any communication between correct processes." [12]. Differently, the values written to a SWMR (single-writer multi-reader) register associated with a non-Byzantine process cannot be corrupted by a Byzantine process. As a result, [16] and [21] considered implementing an array of n SWMR registers, one per process. If a register is associated with some process p, p is the only process that can write it, while all processes can read it.

Defining Byzantine-tolerant registers. Following the definition of Byzantine linearizability proposed in [12], the most natural way of specifying a register in this context would be to say that if the writer is not Byzantine, the register respects the classical specification of an atomic register, otherwise, a read operation can return any value. This register will be called the *Read/Write register*.

In order to give some significance to the registers associated with Byzantine processes, [16] and [21] give a different specification of a SWMR register. Indeed, while the specification of a register associated with a correct (non-Byzantine) process is identical to that of a classical atomic register, the specification of a register associated with a Byzantine process can be declined in different ways, depending on how the behavior of the Byzantine writer is perceived. In [16], a register keeps track of the sequence of all the values written by the writer (be it Byzantine or not) and this sequence is seen in the same way by all non-Byzantine processes. This register will be called in this paper *Read/Append register*. If the writer is correct, the sequence will correspond to the chronological sequence of values it wrote. If the writer is Byzantine, this sequence depends on the behavior of the writer, in the extreme case, the history will be reduced to the initial state (an empty sequence). In other words, a single-writer Read/Append register can be seen as a single-writer atomic ledger.

Differently, in [21], a register keeps only the last written value together with its sequence number. This will be called the *Read/Write-Increment register*. In the same way, if the writer is Byzantine, the register may be in any state as soon as it is perceived consistently by non-Byzantine processes: two correct processes that read the same register and return the same sequence number will necessarily get the same value even if the writer is Byzantine. At the extreme, the register's state will be stuck to the initial value with sequence number 0.

Why only studying SWMR registers? In addition to the lack of sense of multi-writer Read/Write registers discussed above, this paper only considers single-writer registers because we are only interested in memory abstractions that are equivalent to the atomic register in crash-prone systems. Indeed, when only crashes are considered, any of these register specifications can be implemented on top of any other, because writes cannot be concurrent when there is a single writing process. On the other hand, this is not the case for the multi-writer versions of the three register specifications: even in crash-prone systems, a ledger has the synchronization power of consensus and can implement a state machine; and it is easy to see that multi-writer Read/Write-Increment registers have consensus number 2.

In crash-prone systems, basic registers are used as building blocks to construct safe synchronization algorithms (i.e. consensus, stack, queue, etc.) and the additional power is brought either by temporal properties, special hardware instructions, or by randomization. Studying and understanding the exact relationship between single-writer registers in the Byzantine framework would allow a similar approach. For example, using single-writer registers plus additional properties like randomization could help to implement Byzantine-tolerant ledgers. Although the relation of crash-prone atomic registers is well-studied, this is not the case for the Byzantine case.

Hence, this work sheds new light on the construction of Byzantine-prone asynchronous atomic registers. When one wants to implement these SWMR registers over an asynchronous Byzantine message-passing distributed system, $t < n/3$ is necessary and sufficient for all three variants [16]. This is intuitive since the three register specifications differ only in the way they deal with Byzantine writers: Moreover, it is known that in order to implement an atomic register on top of an asynchronous message-passing system prone to process crashes, readers have to write [4, 5]. So even if we only consider non-Byzantine writers, the values

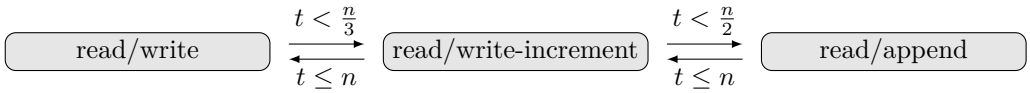


Figure 1 Conditions for implementing Byzantine-tolerant single-writer multi-reader registers.

they write will be relayed by readers which may be Byzantine. Therefore, the correct readers must be able to distinguish the values relayed by Byzantine processes from others hence the ratio $t < n/3$.

Contribution. This paper investigates the relationship between the three register specifications stated above in the presence of Byzantine failures and studies under which condition one type of register can be built from another as shown in figure 1. Whereas the reduction from a Read/Write register to a Read/Write-Increment one and from a Read/Write-Increment register to a Read/Append register is straightforward by their respective definitions, it is important to note that a Read/Write-Increment register can be implemented from Read/Write registers only if $t < n/3$ and surprisingly, a Read/Append register can be reduced to a Read/Write-Increment register as soon as $t < n/2$. This shows that the sequence number mechanism in the R/WI registers is actually quite powerful, but does not close the gap with the R/A registers. This suggests that some aspects of the bad behavior of Byzantine processes are already captured by Read/Write-Increment registers and can benefit upper-layer constructions. The proposed bounds are tight, we prove on the one side that the proposed bounds are necessary, and we propose constructions that allow these reductions.

Organization. The remainder of this paper is organized as follows: Section 2 presents the considered distributed model. Then we investigate the relationships, on the one hand, between Read/Write registers and Read/Write-Increment registers (Section 3) and on the other hand, between Read/Write-Increment registers and Read/Append registers (Section 4). Finally, Section 5 concludes the paper.

2 Computing Model

We consider the classical Byzantine-prone asynchronous shared-memory model.

Computing entities. The system is made up of n sequential processes, denoted $p_1, p_2, \ldots, p_n$. These processes are asynchronous in the sense that each process progresses at its own speed, which can be arbitrary and may vary along any execution, and remains always unknown to the other processes. Each process p_i has access to its own identifier i it can use in the code.

Failure model. A Byzantine process is a process that behaves arbitrarily [18, 22]: it may start in an arbitrary state, stop executing at any time (this behavior is called a crash), perform arbitrary state transitions, attempt to communicate arbitrary or different values to different processes, etc. Among the n processes of the system, it is supposed that at most t can exhibit a Byzantine behavior in any given execution. A Byzantine process is also called a *faulty* process and a process that commits no failure (i.e., a non-Byzantine process) is also called a *correct* process.

Communication model. The processes communicate by invoking operations on a collection of shared objects. Since efficiency is not a central issue of this paper, we consider that processes have access to as many shared objects as necessary. We consider only linearizable

(atomic) t-resilient shared objects, as defined thereafter. t-resiliency is a classical liveness condition stating that all operations invoked by correct processes terminate provided there are at most t faulty processes [7].

Following [12], Definition 1 defines Byzantine linearizability in terms of admitted distributed histories, depending on a given object defined by a sequential specification (in our case, the register specifications already discussed). A distributed history (or simply history when it is clear from context) is an abstraction of a distributed execution, composed of a sequence of atomic steps taken by the processes. The steps taken by correct processes can be either 1) the invocation and response events of the operations specified by a given algorithm, 2) executions of atomic operations on shared objects of the underlying model, or 3) local computations. We only accept well-formed histories, in which correct processes alternate invocation and response events. In other word, a correct process cannot return from an operation it has not yet invoked, and cannot invoke an operation before it has returned from its previous invoked operation.

More precisely, Byzantine linearizability is defined as an extension of linearizability [15], such that only the operations of correct processes must be correctly specified, whereas the histories of Byzantine processes can be re-interpreted into any local history, so that the full history respects the specification of the object. In particular, operations correctly performed by Byzantine processes may be either recognized, ignored or reinterpreted as a different correct operation by the correct processes. A consequence of this definition is that no process can alter the semantics of an operation invoked on a shared object. A Byzantine process can invoke or not an operation independently from its code. However, if Byzantine processes attempt to alter the internal implementation of the object, correct processes will agree on some sequence of operations that were invoked, and the effect of these operations will conform to the specification of the object. This notion of linearizability is close to the one defined in [11] for the special case of the ledger data structure.

▶ **Definition 1** (Byzantine Linearizability). *A history H is* linearizable *with respect to an object O if there exists a sequential history H' (called a linearization of H) such that (1) after removing some pending operations from H and completing others by adding matching responses, it contains the same invocations and responses as H, (2) if an operation o returns before an operation o' starts in H, then o appears before o' in H', and (3) H' satisfies O's sequential specification.*

A history H is Byzantine linearizable with respect to an object O if there exists a history H' linearizable with respect to O, such that $H'|_{correct} = H|_{correct}$ (where $H|_{correct}$ denotes the history H where only the operations done by correct processes are considered).

We say that an object is Byzantine linearizable, or simply linearizable if all of its executions are Byzantine linearizable.

The three register abstractions. This paper considers three variations of the classical SWMR register, whose sequential specification is defined below and illustrated on Figure 2:

Read/Write (R/W) registers offer two operations: a `write` operation, only accessible to the writing process, that does not return any value, and a `read` operation accessible to all processes, that returns the last written value, or the initial value $\perp$ if no value was written.

Read/Write-Increment (R/WI) registers offer two operations: a `write-incr` operation, only accessible to the writing process, that does not return any value, and a `read` operation accessible to all processes, that returns a pair $x = \langle x.\mathsf{value}, x.\mathsf{count}\rangle$, such that

x.value is the last written value (similar to R/W registers), and x.count is the number of times the operation `write-incr` was called by the writer. If no value was ever written, the `read` operation returns $\langle \perp, 0 \rangle$.

Remark that the definition of Byzantine linearizability implies that, if two correct processes read the same count field, then they must read the same value field as well.

Read/Append (R/A) registers offer two operations: an `append` operation, only accessible to the writing process, that does not return any value, and a `read` operation accessible to all processes, that returns the ordered sequence of all written values on that register since the beginning of the execution.

The initial empty sequence is denoted by ε, and the concatenation of a sequence l and a value v is denoted by $l \oplus v$. Given a sequence l and an index $s \in \mathbb{N}$, let us denote by $l[s-1]$ the s^{th} value in l (i.e. the first index is 0). Given two sequences l and l', let $l \subseteq l'$ denote the fact that l is a prefix of l'.

Thanks to Byzantine linearizability, the sequences returned to different reads must be consistent, i.e. one must be a prefix of the other.

```
1 initial state is
2     value ← ⊥;
3 operation write (v) invoked by p_i is
4     value ← v;
5 operation read () invoked by any p_j is
6     return value;
```

(a) Read/write register REG[i].

```
1 initial state is
2     value ← ⊥;
3     count ← 0;
4 operation write-incr(v) invoked by p_i is
5     value ← v;
6     count ← count + 1;
7 operation read() invoked by any p_j is
8     return ⟨value, count⟩;
```

(b) Read/write-inc register REG[i].

```
1 initial state is
2     log ← ε.
3 operation append(v) invoked by p_i is
4     log ← log ⊕ v.
5 operation read() invoked by any p_j is
6     return log.
```

(c) Read/append register REG[i].

Figure 2 Specification of the different registers.

▶ Remark. If a correct process is supposed to write only once in its register, a Byzantine process p_w can write a first value that will be read by some correct processes p_i and then write a second value before some other correct processes p_j reads the register. If the Byzantine process uses a R/W register, p_i and p_j may read different values. If the considered register is a R/WI one, p_j knows that the writing process is Byzantine because the value it reads is associated with a count $= 2$. Finally, if one uses a R/A register, all sequences read by correct processes will contain the very same first value even though the Byzantine process appended other values. It is clear that the different registers offer different information on the behavior of Byzantine processes.

Notation. Let R denote a type of SWMR register through which processes can communicate. The acronym $\mathcal{BASM}_{n,t}[R]$ is used to denote the n-process asynchronous system where up to t processes may exhibit Byzantine behavior and communication is through as many instances of R as necessary. $\mathcal{BASM}_{n,t}[R, C]$ denotes $\mathcal{BASM}_{n,t}[R]$ enriched with the condition C on t and n.

A characterization of Byzantine linearizability for Read/Append registers. In order to simplify the proofs of subsequent algorithms, Proposition 2 defines four properties that characterize linearizable R/A registers. Clearly, these properties are verified by any linearizable R/A register.

▶ **Proposition 2** (Linearizability for Read/Append registers). *Let H be a distributed history of a Read/Append register object that verifies the following properties.*

Validity: *If a* `read` *operation performed by a correct process returns* log*, and if the writing process is correct, then for all $s \in \{1, ..., |log|\}$, $\log[s-1]$ is the s^{th} value written. (By indistinguishability with the scenario where the writer crashes before the end of the read, this implies that the s^{th} write started before the read completed).*

Read after write: *if a read done by a correct process starts after the s^{th} write of a correct process completes, then the read cannot return a sequence containing less than s values.*

Inclusion: *let r_i and r_j be two* `read` *operations, done by correct processes, that return respectively log_i and log_j. Then log_i is a prefix of log_j, or log_j is a prefix of log_i.*

Read after read: *let r_i and r_j be two* `read` *operations, done by correct processes, that return respectively log_i and log_j. If r_i completes before r_j starts, then log_i is a prefix of log_j.*

Then H is Byzantine linearizable with respect to the Read/Append register.

Proof.

Proof when the writing process is correct. Let us consider the history $H' = H|_{correct}$, i.e. H' is H in which the reads of Byzantine processes were removed. Clearly, $H'|_{correct} = H|_{correct}$. We will prove that H' is linearizable.

For each operation o of H', we define the timestamp $ts(o)$ of o as follows. If o is the s^{th} write operation, then $ts(o) = s$. If o is a read operation that returns log, then $ts(o) = |log|$. We also define the binary relation $\rightarrow$ between operations as $o_1 \rightarrow o_2$ if either 1) o_1 returned before o_2 was started (denoted by $o_1 \rightarrow_1 o_2$), or 2) $ts(o_1) < ts(o_2)$ (denoted by $o_1 \rightarrow_2 o_2$), or 3) o_1 is a write, o_2 is a read, and $ts(o_1) = ts(o_2)$ (denoted by $o_1 \rightarrow_3 o_2$).

Let us prove that $\rightarrow$ is cycle-free. Indeed, suppose there is a cycle $o_1 \rightarrow o_2 \rightarrow \ldots \rightarrow o_k = o_1$. Since none of the cases contains reflexivity, the cycle contains at least two different operations. Moreover, let us notice that $o_1 \rightarrow o_2$ implies $ts(o_1) \leq ts(o_2)$: this is true by definition for $\rightarrow_2$ and $\rightarrow_3$, and this is implied by the Validity, Read after read and Read after Writes properties for $\rightarrow_1$. Hence, all operations in the cycle have the same timestamp,

and they are not compared by $\rightarrow_2$. Since $\rightarrow_1$ itself is a partial order, it means there is a write operation w and a read operation r such that $w \rightarrow_3 r$. Consequently, there is a write operation w' and a read operation r' such that $r' \rightarrow w'$, which is only possible if $r' \rightarrow_1 w'$. In other words, a read returns a value that has not yet been written, which is prevented by the Validity property.

Finally, the reflexive and transitive closure of $\rightarrow$ can be extended into a total order $\prec$ that respects real time thanks to $\rightarrow_1$, and that respects the sequential specification of the Read/Append register: by $\rightarrow_2$, the `read` operations are ordered by size of the returned sequences, hence by inclusion by the Inclusion property, and the sequences are composed of written values by the Validity property; `append` operations are ordered before the `read` operations that include their values in their return sequence by $\rightarrow_1$, and after the others by $\rightarrow_2$. Hence, H' is linearizable, so H is Byzantine linearizable, which concludes the proof when the writer is correct.

Proof when the writing process is Byzantine. Let us consider the history $H' = H|_{correct}$, i.e. H' only contains the reads done by the correct processes in H. As is the case of a correct writer, we define the binary relation $\rightarrow$ between two read operations o_1 and o_2, that return respectively log_1 and log_2, as $o_1 \rightarrow o_2$ if either 1) o_1 was terminated before o_2 was started (denoted by $o_1 \rightarrow_1 o_2$), or 2) log_1 is a strict prefix of log_2 (denoted by $o_1 \rightarrow_2 o_2$). The Read after read property implies that $\rightarrow$ is cycle-free, so it can be extended into a total order $\prec$.

Thanks to the Inclusion property and the definition of $\rightarrow_2$, for all operations o_1 and o_2 that return respectively log_1 and log_2, if $o_1 \prec o_2$, then log_1 is a prefix of log_2. Since $\prec$ contains $\rightarrow_2$, it is possible to build a linearizable extension H'' of H' by adding write operations corresponding to the read values, in the order in which they are read, and just before the time where they were read first. Hence, H'' is linearizable and $H''|_{correct} = H|_{correct}$, so H is Byzantine linearizable, which concludes the proof. ◀

3 From R/W registers to R/WI registers

This section shows that the hypothesis $t < \frac{n}{3}$ is necessary, and sufficient, to implement a R/WI register on top of R/W registers. More precisely, Section 3.1 proves that $t < \frac{n}{3}$ is an upper bound on the number Byzantine tolerated by any such reduction algorithm, and then Section 3.2 presents an algorithm whose resilience is optimal.

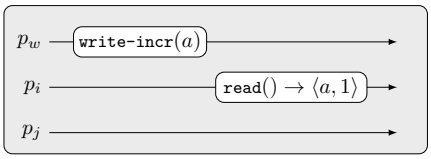
3.1 An upper bound on resilience

This section proves that the maximal resilience of any implementation of a linearizable Read/Write-Increment register in a system where processes communicate only through Read/Write registers, is at most $t \geq \frac{n}{3}$.

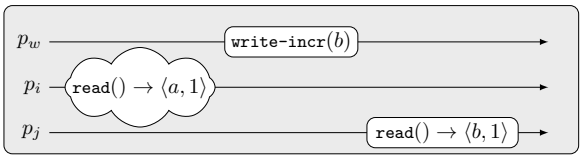
▶ **Theorem 3.** *It is impossible to implement a linearizable Read/Write-Increment register in the model $\mathcal{BASM}_{n,t}[R/W]$, when $n \geq 3$ and $t \geq \frac{n}{3}$.*

Proof. Let us assume that there exists an algorithm, A, which implements linearizable R/WI registers using a collection of R/W registers, even when $t \geq \frac{n}{3}$. We are going to show (proof by contradiction) that there exists an execution allowed by A and that is not linearizable.

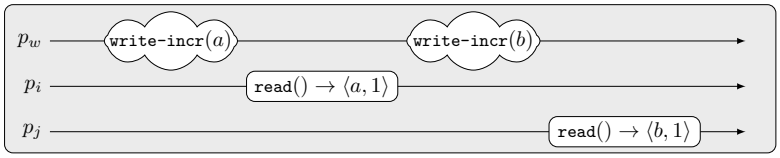
Let us consider a system made up of $n \geq 3$ processes, and let $t \geq \frac{n}{3}$. We can partition the set of processes into three non-empty sub-sets W, I and J, whose size is at most t. Let us pick three processes $p_w \in W$ (the writing process), $p_i \in I$ and $p_j \in J$ (two reading processes), and let us consider three situations in which A is used to implement a R/WI register x that can be written by p_w only. These situations are represented in Figure 3 and described thereafter.



(a) Situation S_1: p_j is Byzantine, but does nothing to prevent p_i from reading $\langle a, 1\rangle$.



(b) Situation S_2: p_i is Byzantine, but cannot prevent p_j from reading $\langle b, 1\rangle$.



(c) Situation S_3: p_w is Byzantine, and can force p_i and p_j to read different values.

Figure 3 Illustration of the scenarios from the proof of Theorem 3, with $n = 3$ and $t = 1$.

S_1: In the first scenario (Figure 3a), the processes in J are Byzantine and do not take any step during the execution. All other processes are correct. Initially, Process p_w writes a in x. The `write-incr` terminates because $|J| \leq t$ processes are Byzantine. Then, Process p_i reads x. Similarly, the `read` terminates, and returns $\langle a, 1\rangle$ because x is linearizable.

S_2: In the second scenario (Figure 3b), only the processes of I are Byzantine. At first, Byzantine processes are not quiet, but they simulate their behavior in S_1. At this point, all shared registers in which processes of I can write are the same as after the reading of $\langle a, 1\rangle$ by p_i. In a second stage, Byzantine processes become quiet and p_w writes b in x. As previously, the `write-incr` must terminate because $|I| \leq t$ processes are Byzantine. In a third stage, p_j reads x. The `read` terminates and returns $\langle b, 1\rangle$ because x is linearizable and the correct process p_w indeed only wrote one value, which was b.

S_3: In the last scenario (Figure 3c), processes of W are Byzantine, and the processes of J are initially slow. Initially, p_w behaves correctly and writes a in x, then Process p_i reads x. So far, the situation is the same as S_1, so p_i gets $\langle a, 1\rangle$ as the result of its read. For the rest of the executions, processes in I become too slow to play any role.

Then, all processes in W write, in their respective registers, the values that were contained in their registers in S_2 after p_w wrote b, and keep simulating the behavior they had in S_2. Finally, p_j reads x. Notice that, at this point, S_3 and S_2 are indistinguishable to all processes in J: in both situations, all registers written by processes in W contain the value resulting from a write of b, all registers written by processes in I contain the value resulting from a write of a, all processes in W respond accordingly to a write of b, and all processes in I are quiet. Therefore, p_j must return $\langle b, 1\rangle$.

Remark that, in S_3, two correct processes p_i and p_j have read respectively $\langle a, 1\rangle$ and $\langle b, 1\rangle$ in x. This violates linearizability, so A cannot exist. ◀

3.2 A resilience-optimal algorithm

Algorithm 1 presents an implementation of a R/A register in the model $\mathcal{BASM}_{n,t}[\mathrm{R/W}, t < \frac{n}{3}]$. Since R/WI registers can be trivially implemented from R/A registers, this proves that R/WI registers can be implemented in $\mathcal{BASM}_{n,t}[\mathrm{R/W}, t < \frac{n}{3}]$ as well. The writing process is denoted by p_w.

Shared memory and local variables. The n processes share three variables called ENDORSE, APPROVE and CONFIRM, defined as follows.

- ENDORSE$[0...][1..n]$ is an infinite array of arrays of n SWMR atomic R/W registers such that, for any $s \in \mathbb{N}$ and $i \in \{1, ..., n\}$, ENDORSE$[s][i]$ can only be written by p_i, is initialized to a value $\perp$ that cannot be appended to the R/A register and eventually contains p_i's opinion on what the s^{th} value appended is.
- Similarly, APPROVE$[0...][1..n]$ is an infinite array of arrays of n SWMR atomic R/W registers, initialized to $\perp$ as well. A correct process p_i only writes, in APPROVE$[s][i]$, a value that it has previously read in more than $\frac{n+t}{2}$ cells of ENDORSE$[s]$.
- CONFIRM$[1..n]$ is an array of n SWMR atomic R/W registers such that for any $i \in \{1, ..., n\}$, CONFIRM$[i]$ can only be written by p_i and contains sequences of appended values that have already been read by p_i in more than $2t$ cells of APPROVE$[s]$.

Besides these three shared variables, each process p_i maintains five local variables:

- log_i is a sequence that represents the current state of the shared R/A register seen by p_i;
- $count_i$ is an integer that represents the number of appended values;
- $endorse_i[1..n]$ is an array of size n, that stores a local copy of ENDORSE$[|log_i|]$ by p_i;
- $approve_i[1..n]$ is an array of size n, that stores a local copy of APPROVE$[|log_i|]$ by p_i;
- $confirm_i[1..n]$ is an array of size n, that stores a local copy of CONFIRM$[|log_i|]$ by p_i.

Algorithm 1 Implementation of a R/A register in the model $\mathcal{BASM}_{n,t}[\text{R/W}, t < \frac{n}{3}]$.

```
   operation append(v) invoked by p_w is
1  |  count_w ← |log_w|;
2  |  ENDORSE[count_w][w].write(v);
3  |  while |log_w| ≤ count_w do synch();

   operation read() invoked by any p_i is
4  |  synch();
5  |  count_i ← max{c ∈ ℕ : |{j : |confirm_i[j]| ≥ c}| > t};
6  |  while |log_i| < count_i do synch();
7  |  return log_i;

   background task T() repeatedly executed by all p_i is
8  |  synch();

   procedure synch() invoked by any p_i is
9  |  for j from 1 to n do endorse_i[j] ← ENDORSE[|log_i|][j].read();
10 |  for j from 1 to n do approve_i[j] ← APPROVE[|log_i|][j].read();
11 |  for j from 1 to n do confirm_i[j] ← CONFIRM[j].read();
12 |  if endorse_i[i] = ⊥ ∧ ∃v ≠ ⊥ : endorse_i[w] = v then ENDORSE[|log_i|][i].write(v);
13 |  if approve_i[i] = ⊥ then
14 |  |  if ∃v ≠ ⊥ : |{j : endorse_i[j] = v}| > (n+t)/2 then APPROVE[|log_i|][i].write(v);
15 |  |  if ∃v ≠ ⊥ : |{j : approve_i[j] = v}| > t then APPROVE[|log_i|][i].write(v);
16 |  else if confirm_i[i] = log_i then
17 |  |  if ∃v ≠ ⊥ : |{j : approve_i[j] = v}| > 2t then CONFIRM[i].write(log_i ⊕ v);
18 |  else if ∃v ≠ ⊥ : |{j : confirm_i[j] ⊇ log_i ⊕ v}| > 2t then log_i ← log_i ⊕ v;
```

Notations. Let v be a value, l a sequence of values, and $s \in \mathbb{N}$. We say that a process p_i *endorses* v as the $(s+1)^{\text{th}}$ value (or simply *endorses* v when s is immaterial) if p_i writes v in ENDORSE$[s][i]$. Similarly, we say that p_i *approves* v as the $(s+1)^{\text{th}}$ value (or simply *approves* v) if p_i writes v in APPROVE$[s][i]$, and that p_i *confirms* l if p_i writes l in CONFIRM$[i]$. We also say that p_i *confirms* v as the $(s+1)^{\text{th}}$ value (or simply *confirms* v) if there exists a sequence l of size s such that p_i confirms $l \oplus v$. Finally, we say that p_i *logs* v as the $(s+1)^{\text{th}}$ value (or simply *logs* v) when p_i executes $log_i \leftarrow log_i \oplus v$ in Line 18, with $|log_i| = s$.

Description of the algorithm. Algorithm 1 implements a confirmation mechanism similar to the protocol for reliable broadcast proposed by Bracha and Toueg in 1985 [8]. In order to write its $(s+1)^{\text{th}}$ value v, the writing process p_w writes v in ENDORSE$[s][w]$ (Line 2) and then waits until it has logged v as its $(s+1)^{\text{th}}$ value, which will happen after enough correct processes have confirmed the write by calling the procedure `synch`() (which they do regularly thanks to Line 8).

When any process p_i invokes `synch`(), it first updates its local copy of the shared variables CONFIRM, APPROVE and ENDORSE (Lines 9–11), and then it checks the five following conditions to help progress on an agreement on the written values.

- The first time p_i reads a non-$\perp$ value in ENDORSE$[s][w]$, it endorses this value by writing it to ENDORSE$[s][i]$ (Line 12).
- If some value v has been endorsed by more than $\frac{n+t}{2}$ processes, i.e. by a majority of the correct processes, p_i approves this value by writing it to APPROVE$[s][i]$ (Line 14).
  Remark that two correct processes cannot approve different values. However, at this stage, it is still possible that some correct process approves a value, and other correct processes do not approve any value.
- If some value v has been approved by at least $t+1$ different processes, i.e. by at least one correct process, p_i trusts this correct process and approves the value as well (Line 15).
- If some value v has been approved by at least $2t+1$ processes, i.e at least $t+1$ correct processes, p_i confirms the value by appending it at the end of CONFIRM$[i]$ (Line 17).
  Remark that if the condition of Line 17 is true for some process, then the condition of Line 15 will always remain true (for the same s) even if t Byzantine processes change their approved value. Hence, all correct processes will eventually confirm the value.
- If some value v has been confirmed by more than $2t$ different processes (correct or not), p_i logs v (Line 18). Recall that $l \sqsupseteq l'$ denotes the fact that l' is a prefix of l.

When any process p_i reads the Read/Append register, it first invokes `synch`() to update its local copies of the shared variables (Line 4), then it computes the number $count_i$ of values that have been confirmed by more than t processes (Line 5) and waits until all these values have been confirmed by at least $2t$ processes (Line 6). This ensures the predicate AIC(log_i) defined by Definition 4: all these values have been confirmed by at least $t+1$ correct processes, which will remain true in the future even if t Byzantine processes change their mind. Then, p_i returns the sequence composed of these values (Line 7) knowing that these values will also be returned by all subsequent reads.

▶ **Definition 4** (Add In Confirm). *For all sequences of values l, let us define the predicate* AIC(l) *as follows:* AIC(l) *is verified if, and only if, there exists at least $t+1$ correct processes p_j such that l is a prefix of* CONFIRM$[j]$.

3.3 Correctness of the Algorithm

We now prove the correctness of Algorithm 1.

▶ **Lemma 5** (Approved value). *Two correct processes cannot approve different s^{th} values.*

Proof. Suppose some correct processes p_i (resp. p_j) writes v_i (resp. v_j) in APPROVE$[s]$, for some s. Since this can happen on Line 15 only if p_i (resp. p_j) has read v_i (resp. v_j) from a correct process in APPROVE$[s]$, some correct process wrote v_i (resp. v_j) in APPROVE$[s]$ on Line 14. By the condition on Line 14, more than $\frac{n+t}{2}$ processes endorsed v_i (resp. v_j). Therefore, at least $t+1$ processes endorsed both values on Line 12, among which there is one correct process. This contradicts the condition $endorse_i[i] = \perp$ of Line 12. ◀

▶ **Lemma 6** (Confirmed value). *If a correct process p_i logs v on Line 18, then it confirmed $log_i \oplus v$.*

Proof. Suppose a correct process p_i writes $log_i \oplus v$ on Line 18. The condition on Line 16 was false, so p_i wrote $log_i \oplus v'$ to CONFIRM$[i]$ on Line 17, for some v' that was approved by a correct process. By the condition on Line 18, some correct process confirmed v on Line 17, so some correct process approved v. Since v and v' were both approved by a correct process, $v = v'$ by Lemma 5. ◀

▶ **Lemma 7** (Logged value). *If the writing process p_w is correct and some correct process p_i logs v as its s^{th} value, then v is the s^{th} value written by p_w.*

Proof. When p_i logs v as its s^{th} value (Line 18), $log_i \oplus v$ has already been confirmed by at least $2t+1$ processes, hence by some correct process in Line 17. Therefore, v was approved by at least $2t+1$ processes, on Line 14 or 15. Remark that the first correct process that approved v could not have done so on Line 15 because, at that moment, it was approved by at most t Byzantine processes. Hence, some correct process approved v on Line 14, after v had been endorsed by more than $\frac{n+t}{2} \geq 2t$ processes. Some correct process among them endorsed v on Line 12. By Line 12, v' is the value v that p_w wrote on Line 2 during its s^{th} invocation of `append`. ◀

▶ **Lemma 8** (Stability of AIC). *If AIC(l) is true for some l at some time, then it remains true forever.*

Proof. Suppose that l is a prefix of CONFIRM$[i]$ for some correct process p_i. If CONFIRM$[i]$ is a prefix of log_i, then it will remain true because p_i only appends values at the end of log_i, to overwrite either log_i or CONFIRM$[i]$. Otherwise, p_i already executed Line 17, but not yet Line 18, and by Lemma 6, it can only write CONFIRM$[i]$ in log_i. Hence, l remains a prefix of CONFIRM$[i]$ forever, and the same is true for all correct processes. ◀

▶ **Lemma 9** (Inclusion for AIC). *If there exist l and l' such that the condition AIC(l) is true at some time, and AIC(l') is true at some time, then l is a prefix of l' or l' is a prefix of l.*

Proof. By Lemma 8, AIC(l) and AIC(l') remain true forever after some point. Then, some correct processes p_i and p_j confirmed l and l', respectively. Suppose (by contradiction) that the lemma is false, and let us consider the longest common prefix log of l and l', as well as the first value $v_i \neq v_j$ by which l and l' differ. As correct processes append values one by one on Line 17, p_i (resp. p_j) confirmed $log \oplus v_i$ (resp. $log \oplus v_j$). By the condition of Line 17, some correct process approved v_i (resp. v_j), which contradicts Lemma 5. ◀

▶ **Lemma 10** (Log and AIC). *If p_i is a correct process, AIC(log_i) holds at all times.*

Proof. We prove the lemma by induction on $|log_i|$. Initially, AIC(ε) holds. Suppose AIC(log_i) holds for some log_i, and let us suppose p_i logs v on Line 18. By the condition of that line, $log_i \oplus v$ was confirmed by at least $2t+1$ processes, hence by at least $t+1$ correct processes. Therefore, AIC($log_i \oplus v$) holds at that time, and by Lemma 8, AIC($log_i \oplus v$) remains true forever afterward. ◀

▶ **Lemma 11** (Reads and AIC). *Let l be a sequence such that AIC(l) is true when a correct process p_i invokes a* `read`() *operation. Then l is a prefix of the sequence returned to p_i.*

Proof. By Line 5, $|l| \leq count_i$, so by Line 6, $|l| \leq |log|_i$. Moreover, AIC(log_i) holds by Lemma 10. Since AIC(l) and AIC(log_i) are satisfied, Lemma 9 implies that l is a prefix of log_i. ◀

▶ **Lemma 12** (Linearizability). *Let H be a distributed history admitted by Algorithm 1. Then H is Byzantine linearizable with respect to the Read/Append register.*

Proof. Following the characterization of Proposition 2, we prove the four properties that imply Byzantine linearizability.

Proof of the Validity property. Correct processes p_i return log_i on Line 7, which is composed of logged values that have been written by p_w according to Lemma 7.

Proof of the Read after Write property. Let us suppose p_w is correct and completed its s^{th} append operation (of some value v) before a correct process p_i starts reading. By Lemma 10, AIC(log_w) holds on Line 3, with $|log_w| \geq s$. By Lemma 8, AIC(log_w) is still true when p_i starts its read, so by Lemma 11, log_w is a prefix of the sequence returned by p_i, of size at least s.

Proof of the Inclusion property. Let us consider two reads r_i and r_j, done by two processes p_i and p_j, that return respectively l_i and l_j. By Lemma 10, we have AIC(l_i) (resp. AIC(l_j)) when p_i (resp. p_j) executes Line 7, which implies the inclusion property by Lemma 9.

Proof of the Read after read property. Let r_i and r_j be two `read` operations done by correct processes p_i and p_j, that return respectively l_i and l_j, such that r_i completes before r_j starts. By Lemmas 8, 10, and 11, AIC(l_i) is verified when p_i returns, then when p_j starts its read, so l_i is a prefix of l_j. ◀

▶ **Lemma 13** (Liveness). *If some correct process p_i confirms a sequence of size s, all correct processes eventually log a s^{th} value.*

Proof. Suppose some correct process p_i confirms a sequence of size s, and some process p_j never writes a sequence of size s in log_j. Since p_j appends values one by one in log_j, this means that $|log_j|$ plateaus at a size $s' < s$ after some point in time. Without loss of generality, let us assume that s' is minimal, i.e. $|log_k|$ reaches at least s' for all correct process p_k. By Lemma 6 and the condition on Line 17, when $|log_i|$ reaches $s'+1$, at least $t+1$ approved $s'+1$ values.

Since all correct processes p_j repeatedly execute `synch`() thanks to Line 8, all processes eventually read v at least $t+1$ times in APPROVE[s'] (and only v, by Lemma 5) and approve it (Line 15). This eventually satisfies the condition on Line 17, and then the condition on Line 18. Hence, $|log_j|$ reaches $s'+1$. A contradiction. ◀

▶ **Lemma 14** (t-resilience). *Algorithm 1 is t-resilient.*

Proof.

Termination of the append operation. Suppose a correct process p_w invokes `append` (v) to write its s^{th} value, and let l_w be log_w on Line 1. By Lemma 10 AIC(l_w) holds at the beginning of the execution. Hence, some correct process p_i writes l_w in CONFIRM$[i]$, and by Lemma 13, each correct process p_i eventually reach a state where $|log_i| = |l_w|$.

Since all correct processes repeatedly execute `synch`() thanks to Line 8, all processes eventually read v in ENDORSE$[|l_w|][w]$ (Line 9) and write it in ENDORSE$[|l_w|][i]$ (Line 12). Since there are at least $n - t > \frac{2n}{3} > \frac{n+t}{2}$ correct processes, eventually, all of them write v in APPROVE$[|l_w|]$ (Line 15). Since there are at least $n - t > \frac{2n}{3} > 2t$ correct processes, eventually, each of them appends v at the end of CONFIRM (Line 17). Then, p_w can read v in the end of CONFIRM$[j]$ (Line 11) for more than $2t$ different j, and appends it to log_w (Line 18) and terminates its while loop (Line 3).

Termination of the read operation. Suppose a correct process p_i invokes `read`(), and let us study its loop on Line 6. By Line 5, at least one correct p_j process wrote a log containing at least $count_i$ values in CONFIRM$[j]$. By Lemma 13, p_i eventually writes at least $count_i$ values in log_i and complete its `read` operation. ◀

▶ **Theorem 15** (Correctness of Algorithm 1)**.** *Algorithm 1 implements a t-resilient linearizable SWRM Read/Append register in the model $\mathcal{BASM}_{n,t}[R/W, t < \frac{n}{3}]$.*

Proof. By Lemma 12, Algorithm 1 is linearizable. By Lemma 14, Algorithm 1 is t-resilient. ◀

4 From R/WI registers to R/A registers

This section shows that the hypothesis $t < \frac{n}{2}$ is necessary, and sufficient, to implement a R/A register on top of R/WI registers. More precisely, Section 4.1 proves that any such reduction algorithm has a resilience $t < \frac{n}{2}$ as an upper bound, and then Section 4.2 presents an algorithm with an optimal resilience.

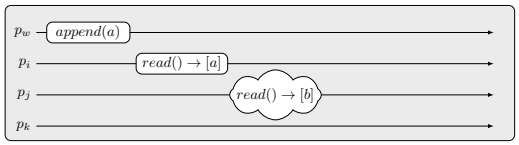
4.1 An upper bound on resilience

This section proves that for any implementation of a linearizable Read/Append register in a system where processes communicating only through Read/Write-Increment registers, the maximal resilience is at most $t < \frac{n}{2}$.

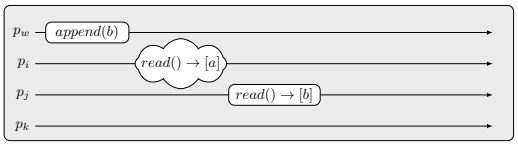
▶ **Theorem 16.** *It is impossible to implement a linearizable Read/Append register in the model $\mathcal{BASM}_{n,t}[R/WI]$, when $n \geq 4$ and $t \geq \frac{n}{2}$.*

Proof. Similarly to Section 3.1, suppose there is an Algorithm A implementing a R/A register from a collection of R/WI registers in a system made up of $n \geq 4$ processes, from which $t \geq \frac{n}{2}$ maybe Byzantine. We will prove, by contradiction, that A allows a run that is not linearizable. Since $n \geq 4$, processes can be divided into four non-empty subsets W, I, J and K, each having a size not exceeding $\frac{t}{2}$ processes. Let us pick four processes $p_w \in W$ (the writing process), $p_i \in I$, $p_j \in J$ and $p_k \in K$ (three reading processes). We will consider the four situations represented in Figure 4 and described thereafter.

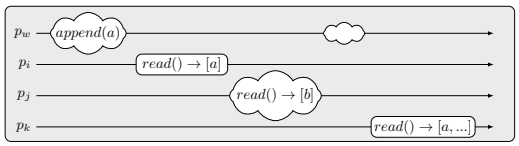
S1: In this first scenario, all processes of the sets J are Byzantine whereas W, I and K consist of correct processes. Processes of K are too slow to participate. Process p_w appends a to x. The write terminates because $|J| \leq t$ processes are Byzantine. Then p_i reads x and obtains $[a]$ because x is linearizable. After that, the Byzantine processes in J will simulate the steps they would have taken in a read of x by p_j, if p_w had appended b instead of a.



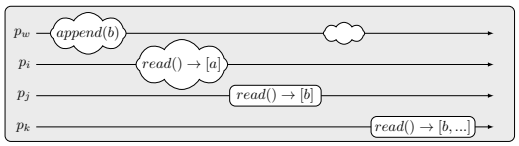
(a) Scenario S_1: p_j and p_k are Byzantine, but cannot prevent p_i from reading $[a]$.



(b) Scenario S_2: p_i and p_k are Byzantine, but cannot prevent p_j from reading $[b]$.



(c) Scenario S_3: p_w and p_j are Byzantine, but cannot prevent p_i and p_j from reading $[a, ...]$.



(d) Scenario S_4: p_w and p_i are Byzantine, but cannot prevent p_j and p_k from reading $[b, ...]$.

Figure 4 Illustration of the scenarios from the proof of Theorem 16, with $n = 4$ and $t = 2$.

S2: In the second scenario, the processes of I are byzantine while the others are correct. Processes of K are still slow. Process p_w appends b to x, then the Byzantine processes of I simulate the steps they would take in a read of x by p_i if the written value were a. Finally, the correct process p_j reads the contents of x and gets $[b]$ as a result because x is linearizable.

For all R/WI registers x, let $x.\mathsf{count}_1$ (resp $x.\mathsf{count}_2$) the value of $x.\mathsf{count}_1$ at the end of S_1 (resp. S_2). The two remaining scenarios are built as extensions of S_1 and S_2, except that processes of W are Byzantine as well, although p_w appends its value (a in S_3 and b in S_4) by following A properly.

S3: After p_i and p_j finish their read in S1, all processes $p'_w \in W$ follow the following strategy to confuse correct processes. For all R/WI registers x on which p'_w can write, and such that $x.\mathsf{count}_1 + x.\mathsf{count}_2 \neq 0$, p'_w calls `write-incr`$(\perp)$ on x until the value of x is $\langle \perp, \max(x.\mathsf{count}_1, x.\mathsf{count}_2) + 1 \rangle$.

Then, process p_k reads the contents of x. Since the correct process p_i already read $[a]$, p_k is forced to return a sequence whose first value is a, because x is linearizable.

S4: Processes $p'_w \in W$ continue S2 with the same strategy. For all R/WI registers x p'_w can write such that $x.\mathsf{count}_1 + x.\mathsf{count}_2 \neq 0$, p'_w calls `write-incr`$(\perp)$ on x until the value of x is $\langle \perp, \max(x.\mathsf{count}_1, x.\mathsf{count}_2) + 1 \rangle$.

Then, process p_k reads a sequence starting with b in x, because x is linearizable and the correct process p_j already read $[b]$.

The contradiction comes from the fact that the scenarios S_3 and S_4 are indistinguishable to process p_k during its last read: all the registers that can be written by processes of W are in the same state that exposes no relevant information, all processes of I pretend they have read a, and all processes of J pretend they have read b. Therefore, it is impossible for p_k to return a different value in S_3 and S_4, which means A cannot exist. ◀

4.2 A resilience-optimal algorithm

Algorithm 2 presents an implementation of a R/A register in the model $\mathcal{BASM}_{n,t}[\text{R/WI}, t < \frac{n}{2}]$. The writing process is denoted by p_w.

Shared memory and local variables. The n processes share two variables called ENDORSE and COUNTER, defined as follows.

Algorithm 2 Implementation of a R/A register in the model $\mathcal{BASM}_{n,t}[\text{R/WI}, t < \frac{n}{2}]$.

```
   operation append(v) invoked by p_w is
1      count_w ← |log_w|;
2      ENDORSE[count_w][w].write-incr(v);
3      COUNTER.write-incr(⊥);
4      while |log_w| ≤ count_w do synch();

   operation read() invoked by any p_i is
5      count_i ← COUNTER.read().count;
6      do old_i ← log_i; synch(); while |log_i| < count_i ∧ old_i ≠ log_i;
7      return log_i;

   background task T() repeatedly executed by all p_i is
8      synch()

   procedure synch() invoked by any p_i is
9      if COUNTER.read().count ≤ |log_i| then return;
10     for j from 1 to n do endorse_i[j] ← ENDORSE[|log_i|][j].read() ;
11     if endorse_i[i].count = 0 ∧ endorse_i[w].count = 1 then
12         ENDORSE[|log_i|][i].write-incr(endorse_i[w].value)
13     if ∃v : |{j : endorse_i[j] = ⟨v, 1⟩ ∨ endorse_i[j].count > 1}| > t then
14         log_i ← log_i ⊕ v;
```

- ENDORSE[0...][1..n] plays the same role as in Algorithm 1: it is an infinite array of arrays of n SWMR atomic R/WI registers such that, for any $s \in \mathbb{N}$ and $i \in \{1, ..., n\}$, ENDORSE$[s][i]$ is initialized to $\langle \perp, 0 \rangle$, can only be written by p_i, and eventually contains p_i's opinion on what the s^{th} appended value is.
  Each process p_i is only supposed to write once in ENDORSE$[s][i]$, for each s. Hence, a Byzantine process can erase a value that it has already written in ENDORSE$[s][i]$, but doing so passes the count field to 2, which informs the other processes that it is faulty and its values cannot be trusted.
- COUNTER is an SWMR atomic R/WI register, initialized to $\langle \perp, 0 \rangle$ as well, and can only be written by the writing process p_w. Only the field count of COUNTER is used to represent the number of appended values, so p_w only writes a dummy value $\perp$ in it.

Besides these two shared variables, each process p_i maintains four local variables:

- log_i is a sequence that represents the current state of the shared R/A register seen by p_i.
- old_i is a local copy of log_i, used to detect when the value of log_i changes locally.
- $count_i$ is an integer that represents the number of appended values.
- $endorse_i[1..n]$ is an array of size n, that stores a local copy of ENDORSE$[|log_i|]$ by p_i.

Notations. Let v be a value and $s \in \mathbb{N}$. We say that a process p_i *endorses* v as the $(s+1)^{\text{th}}$ value (or simply *endorses* v when s is immaterial) if the first value p_i writes in ENDORSE$[s][i]$ is v. Similarly, we say that p_i *logs* v as the $(s+1)^{\text{th}}$ value (or simply *logs* v) when p_i executes $log_i \leftarrow log_i \oplus v$ in Line 18, with $|log_i| = s$.

Description of the algorithm. Similarly to Algorithm 1, Algorithm 2 implements a confirmation mechanism to ensure that a read value can never be lost. In order to write its $(s+1)^{\text{th}}$ value v, the writing process p_w writes v in ENDORSE$[s][w]$ (Line 2), then increments

the count field of COUNTER (Line 3) by calling `write-incr`($\perp$), and then waits until it has logged at s values. This happens eventually after enough correct processes have endorsed v as their $(s+1)^{\text{th}}$ value by calling the procedure `synch`() (which they do regularly on Line 8).

When any process p_i invokes `synch`(), it first checks whether a new value was appended by comparing the count field of COUNTER, to the length of its log_i variable (Line 9), and it updates its local copy of the shared variable ENDORSE (Line 10). Then on Lines 11–12, p_i endorses the value $endorse_i[w]$.value as its $(|log_i|+1)^{\text{th}}$ value if 1) it has not done so yet (i.e. $endorse_i[i]$.count $= 1$), and 2) p_w has endorsed this value and has not overwritten it (i.e. $endorse_i[w]$.count $= 1$).

Finally, p_i logs v if enough correct processes have endorsed the same value v to ensure its persistence (Lines 13–14). The condition for persistence, captured by the predicate AIH$(|log_i|, v)$ in Definition 17, is similar to the condition of Line 13, and has two important properties:

- It can only be true if v was endorsed by some correct process, hence if v was endorsed by the writer itself. Therefore, only one value v can ever satisfy this property
- Once this property is true at the same process, it cannot be falsified at another process later.

When any process p_i reads the Read/Append register, it first sets its local variable $count_i$ to the count field of COUNTER, which indicates the number of written values if the writer is correct, and a bound on the complexity of the operation otherwise. Then, it invokes `synch`() repeatedly until either 1) it has read $count_i$ values, or 2) its value for log_i does not change during one iteration (which indicates that a Byzantine writer incremented COUNTER without updating ENDORSE properly). Then, p_i returns its local value for log_i.

▶ **Definition 17** (Add In History). *For all $s \in \mathbb{N}$ and all values v, let $\text{AIH}(s, v)$, be the predicate $\text{AIH}_1(s, v) \wedge \text{AIH}_2(s, v)$ defined as follows:*

- $\text{AIH}_1(s, v) \triangleq \mathit{COUNTER}.\mathsf{count} > s$,
- $\text{AIH}_2(s, v) \triangleq |\{i : \mathit{ENDORSE}[s][i] = \langle v, 1\rangle \vee \mathit{ENDORSE}[s][i].\mathsf{count} > 1\}| > t$.

4.3 Correctness of the Algorithm

We now prove the correctness of Algorithm 2.

▶ **Lemma 18** (Stability of AIH). *If the condition $\text{AIH}(s, v)$ is true at a given time for some pair (s, v), it can never become false afterward.*

Proof.

Proof for aih_1. COUNTER.count cannot decrease by the definition of write-increment.

Proof for aih_2. Let p_i be a process such that ENDORSE$[s][i] = \langle v, 1\rangle \vee$ ENDORSE$[s][i]$.count > 1. If p_i does not overwrite ENDORSE$[s][i]$, ENDORSE$[s][i]$ keeps its value because it is a SWMR register, and if p_i writes in ENDORSE$[s][i]$ afterwards, it remains true that ENDORSE$[s][i]$.count > 1. ◀

▶ **Lemma 19** (Safety of AIH). *Suppose the condition $\text{AIH}(s, v)$ is true at a given time for some pair (s, v). Then at some point in the execution, $\mathit{ENDORSE}[s][w] = \langle v, 1\rangle$.*

Proof. For $\text{AIH}_2(s)$ to be true, at least $t+1$ processes p_i must have written in ENDORSE$[s][i]$. Among them, some process p_j must be correct. If p_j is the writer p_w, then the write happened on Line 2, and after that ENDORSE$[s][w] = \langle v, 1\rangle$. Otherwise, the write happened on Line 12. By the condition on Line 11, p_j read $\langle v, 1\rangle$ in ENDORSE$[s][w]$ on Line 10, which concludes the proof. ◀

▶ **Lemma 20** (Update of log_i). *Let us consider a call of* synch() *by any correct process p_i, let s be $|log_i|$ at the moment of the invocation, and let v be any value. Then:*

- *if* AIH(s, v) *when p_i invokes* synch(), *then p_i appends v on Line 14;*
- *if p_i appends v on Line 14, then* AIH(s, v) *is true when p_i returns from* synch().

Proof. Suppose AIH(s, v) when p_i invokes synch(). Then p_i does not return on Line 9 by AIH$_1(s, v)$, and AIH$_2(s, v)$ is true when p_i executed Line 10, so the condition on Line 13 is true.

Suppose p_i appends v on Line 14. Then AIH$_1(s, v)$ was true when p_i executed Line 9, and AIH$_2(s, v)$ was true after Line 10. Hence, Lemma 18 concludes the proof. ◀

▶ **Lemma 21** (Linearizability). *Let H be a distributed history admitted by Algorithm 2. Then H is Byzantine linearizable with respect to the Read/Append register.*

Proof. Following the characterization of Proposition 2, we prove the four properties that imply Byzantine linearizability.

Proof of the Validity property. Suppose the writing process p_w is correct, let us consider the sequence log_i returned by the read of a correct process p_i on Line 7, let $s \in \{1, ..., |log|\}$, and let $v = log[s-1]$. By Lemma 20, AIH$(s-1, v)$ is true after p_i appended v to log, so by Lemma 19, ENDORSE$[s-1][w] = \langle v, 1\rangle$. Since p_w is correct, p_w executed Line 2 when it wrote v as its s^{th} value.

Proof of the Read after Write property. Let us suppose p_w is correct and completed its s^{th} write (of some value v) before a correct process p_i starts reading. By Line 4, the write can only stop when $|log_w| = s$, so by Lemma 20, AIH$(s-1, v)$ is true at the end of the write. By Lemma 18, AIH$(s-1, v)$ is still true at the beginning of the read. By the same reasoning for previous writes (that have been completed as well), AIH$(s', _)$ is true for all $s' < s$. By AIH$_1(s-1, v)$, $count_i \geq s-1$ after Line 5, and by Lemma 20, some value is appended to log_i each time synch() is called on Line 6 when $|log_i| < s$. Hence, $|log_i| \geq s$ when Line 6 completes.

Proof of the Inclusion property. The sequence returned by a correct process is the content of its variable log_i. By Lemmas 20 and 19, all processes update their log_i value by appending the same values in the same order.

Proof of the Read after read property. Let r_i and r_j be two read operations done by correct processes p_i and p_j, that return respectively log_i and log_j, such that r_i completes before r_j starts. By Lemma 20, AIH$(s', _)$ is true for all $s' < |log_i|$ at the end of r_i. Hence, applying the same reasoning as for the Read after write property, $|log_j| \geq |log_i|$. Finally, the Inclusion property implies that log_i is a prefix of log_j. ◀

▶ **Lemma 22** (t-resilience). *Algorithm 1 is t-resilient.*

Proof.

Termination of the append operation. Suppose a correct process p_w invokes append(v) to write its $(s+1)^{th}$ value. In particular, p_w completed all previous writes, by Lemma 20, AIH$(s', _)$ is true for all $s' < s$, and AIH$_1(s, v)$ is true by Line 3.

All correct processes repeatedly execute synch() thanks to Line 8, hence by Lemma 20 again, for all correct processes p_i, eventually $|log_i| = s-1$. The next time p_i executes synch(), it reads v in ENDORSE$[s-1][w]$ (Line 10) and writes it in ENDORSE$[s-1][i]$ (Line 12), which is enough to satisfy AIH$_2(s, v)$ and allow p_w to complete its write.

Termination of the read operation. Suppose a correct process p_i invokes `read`(). Since log_i can only be updated by appending values at its end (Line 14), at each iteration of the loop Line 6, either log_i remains unchanged which stops the loop, or the length of log_i growths, until it reaches $count_i$ which stops the loop as well. ◀

▶ **Theorem 23** (Correctness of Algorithm 2). *Algorithm 2 implements a t-resilient linearizable SWRM R/A register in the model* $\mathcal{BASM}_{n,t}[R/WI, t < \frac{n}{2}]$.

Proof. This is a direct consequence of lemmas 21 and 22. ◀

5 Conclusion

The goal of this paper is to investigate the relationships between three register specifications: the Read/Write register, whose `read` operation returns the last written value, the Read/Write-Increment register, whose `read` operation returns a pair composed of the last written value and the total number of values written, and the Read/Append register, whose `read` operation returns the sequence of all written values. We identified necessary and sufficient bounds on the number of Byzantine failures that can be tolerated in algorithms that build one from another. More precisely, a Read/Write-Increment register can be implemented on top of Read/Write registers if, and only if, $t < \frac{n}{3}$. Differently, a Read/Append register can be implemented on top of Read/Write-increment registers at the condition that $t < \frac{n}{2}$.

In order to prove that Read/Write-Increment registers can be implemented on top of Read/Write registers when $t < \frac{n}{3}$, Algorithm 1 actually provides an implementation of a Read/Append register. This is correct for computability reasons because Read/Write-Increment registers can be trivially obtained from Read/Append registers. However, this poses the question of the memory complexity of such algorithms: it is expected that a Read/Append register needs to keep the entire sequence of written values because of its specification, but is it possible to only keep track of the current value and a sequence number in a Read/Write-Increment register implementation?

References

1 Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Comput.*, 18(5):387–408, 2006. doi:10.1007/S00446-005-0151-6.

2 Ittai Abraham and Dahlia Malkhi. The blockchain consensus layer and BFT. *Bull. EATCS*, 123, 2017. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/506.

3 Amitanand Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 310–311, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1281100.1281147.

4 Hagit Attiya. Efficient and robust sharing of memory in message-passing systems. *J. Algorithms*, 34(1):109–127, 2000. doi:10.1006/JAGM.1999.1025.

5 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.

6 Hagit Attiya and Amir Bar-Or. Sharing memory with semi-byzantine clients and faulty storage servers. *Parallel Process. Lett.*, 16(4):419–428, 2006. doi:10.1142/S0129626406002745.

7 E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. of the 25th Annual ACM Symposium on Theory of Computing STOC*, pages 91–100, 1993. doi:10.1145/167088.167119.

8 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985. doi:10.1145/4221.214134.

9 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.

10 Gregory V. Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distributed Comput.*, 18(1):73–84, 2005. doi:10.1007/S00446-005-0123-X.

11 V. Cholvi, A. Fernández Anta, C. Georgiou, N. Nicolaou, and M. Raynal. Atomic appends in asynchronous byzantine distributed ledgers. In *16th European Dependable Computing Conference EDCC*, pages 77–84. IEEE, 2020. doi:10.1109/EDCC51268.2020.00022.

12 Shir Cohen and Idit Keidar. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.DISC.2021.18.

13 Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. The complexity of robust atomic storage. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 59–68, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993806.1993816.

14 Rachid Guerraoui and Marko Vukolić. How fast can a very robust read be? In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 248–257, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1146381.1146419.

15 Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

16 Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. *J. Parallel Distributed Comput.*, 93-94:1–9, 2016. doi:10.1016/J.JPDC.2016.03.012.

17 Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Comput.*, 1(2):77–85, 1986. doi:10.1007/BF01786227.

18 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.

19 Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in phalanx. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, page 51, USA, 1998. IEEE Computer Society. doi:10.1109/RELDIS.1998.740474.

20 Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic byzantine storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 325–334, USA, 2004. IEEE Computer Society. doi:10.1109/DSN.2004.1311902.

21 Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory Comput. Syst.*, 60(4):677–694, 2017. doi:10.1007/S00224-016-9699-8.

22 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

23 Fred B. Schneider. The state machine approach: A tutorial. In *Proc. of Asilomar Workshop on Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 18–41. Springer, 1986. doi:10.1007/BFB0042323.