Two for One, One for All: Deterministic LDC–Based Robust Computation in Congested Clique

Keren Censor-Hillel
Technion, Haifa, Israel

Orr Fischer
Bar-Ilan University, Ramat Gan, Israel

Ran Gelles
Bar-Ilan University, Ramat Gan, Israel

Pedro Soto
Virginia Tech, Blacksburg, VA, USA

Abstract

We design a deterministic compiler that makes any computation in the Congested Clique model robust to a constant fraction $\alpha < 1$ of adversarial crash faults. In particular, we show how a network of n nodes can compute any circuit of depth d, width ω, and gate total fan Δ, in $d \cdot \lceil \frac{\omega}{n^2} + \frac{\Delta}{n} \rceil \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds in such a faulty model. As a corollary, any T-round Congested Clique algorithm can be compiled into an algorithm that completes in $T^2 n^{o(1)}$ rounds in this model.

Our compiler obtains resilience to node crashes by coding information across the network, and its main underlying observation is that we can leverage locally-decodable codes (LDCs) to maintain a low complexity overhead, as these allow recovering the information needed at each computational step by querying only small parts of the codeword, instead of retrieving the entire coded message, which is inherent when using block codes.

The main technical contribution is that because erasures occur in known locations, which correspond to crashed nodes, we can *derandomize* classical LDC constructions by deterministically selecting query sets that avoid sufficiently many erasures. Moreover, when decoding multiple codewords in parallel, our derandomization *load-balances* the queries per-node, thereby preventing congestion and maintaining a low round complexity.

Deterministic decoding of LDCs presents a new challenge: the adversary can target precisely the (few) nodes that are queried for decoding a certain codeword. We overcome this issue via an adaptive doubling strategy: if a decoding attempt for a codeword fails, the node doubles the number of its decoding attempts. We employ a similar doubling technique when the adversary crashes the decoding node itself, replacing it dynamically with two other non-crashed nodes. By carefully combining these two doubling processes, we overcome the challenges posed by the combination of a deterministic LDC with a worst case pattern of crashes.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Congested Clique, Fault Tolerance, Error Correction Codes

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.20

Related Version *Full Version*: https://doi.org/10.48550/arXiv.2508.08740 [8]

Funding *Keren Censor-Hillel*: is supported in part by the Israel Science Foundation, grant 529/23. *Orr Fischer*: is supported in part by the Israel Science Foundation, grant No. 1042/22 and 800/22. *Ran Gelles*: supported in part by the United States – Israel Binational Science Foundation (BSF), grant No. 2020277.

Acknowledgements We would like to thank Merav Parter and Noga Ron-Zewi for helpful discussions.

1 Introduction

Robustness is a crucial component in the design of distributed algorithms, as faults lie at the heart of distributed computing environments. Thus, addressing various types of failures has been heavily studied, see, e.g., seminal results covered in classic books on distributed computing [2, 37, 44]. In this paper, we focus on *node crashes* in the Congested Clique model (introduced by Lotker et al. [36]), in which the computing devices use bandwidth-restricted point-to-point communication over a complete network graph. So far, despite abundant research in this model (see related work in Section 1.2), relatively less attention was given to fault tolerance questions in this useful model [3, 15, 24, 33, 34].

In some settings, such as the Congest model or the work in Congested Clique of [33, 34], if node crashes are allowed then the requirement is that the output of the computation corresponds only to the inputs of non-crashed nodes. In contrast, Censor-Hillel and Soto [13] show how to avoid losing any information in the Congested Clique despite node crashes. They do this by having each node encode its input and the results of any subsequent local computations using erasure correction codes, split the codewords to pieces and distribute them to the nodes of the network. Upon a node crash, the other nodes collect the pieces of the relevant codeword and decode it in order to continue the computation. This approach implies that any task over a network of n nodes (with the typical setup of $O(n \log n)$ bits of input per node which we will also work with here) can be computed in $O(n)$ rounds despite crashes: after each node distributes its inputs in an encoded way, all other nodes gather all the pieces and reconstruct the entire $O(n^2 \log n)$-bit input, from which they can locally compute the output. The work [13] beats this bound for certain tasks whose circuit representation has good properties. While their work can achieve fast resilient algorithms for well-behaved circuits, e.g., retain the $\tilde{O}(n^{1/3})$-round complexity for matrix multiplication [12] even with faults, for a *general circuit*, their resilient algorithm may incur a multiplicative overhead of n rounds, which is worse than the solution that simply learns the entire encoded input.

In this paper, we provide a compiler that makes Congested Clique algorithms robust to a constant fraction of node crashes, by computing general circuits faster. This results in a compiler with a complexity of $T^2 n^{o(1)}$ rounds for a Congested Clique algorithm of T rounds, and thus it beats the $O(n)$-round solution when $T = o(n^{1/2-o(1)})$. In the crash model we consider, node crashes occur at the start of a round, and the adversary may crash up to αn nodes in total throughout the algorithm, for a fault parameter $\alpha \in [0, 1)$.

▶ **Theorem 1.** *Let* ALG *be any Congested Clique algorithm, such that each node's input and randomness string is of size at most $\widetilde{O}(n)$ bits, and that completes in T rounds. Then, for any $\alpha \in [0, 1)$, there is an equivalent algorithm* ALG′ *that is resilient to an α fraction of crashes and completes in $T^2 n^{o(1)}$ rounds.*

A concrete example for an application of Theorem 1 is computing exact single-source-shortest-paths (SSSP) in weighted undirected graphs. The $\tilde{O}(n^{1/6})$-round algorithm of [6] translates by our compiler to an algorithm that completes in $n^{1/3+o(1)}$ rounds, even if any αn nodes may crash during its execution, for any $\alpha < 1$.

At the heart of our compiler is a faster deterministic algorithm for computing general circuits in the faulty model. The following is our main technical contribution.

▶ **Theorem 2.** *Let C be circuit of depth d, max total-fan Δ, and width ω. Then, for any $\alpha \in [0, 1)$ there exists a deterministic Congested Clique algorithm for computing the output of C in the presence of αn crashes, whose round complexity is $d \cdot \lceil \frac{\omega}{n^2} + \frac{\Delta}{n} \rceil \cdot 2^{O(\sqrt{\log n} \log \log n)}$.*

While our transformation also goes through computing circuits, our algorithm differs from that of [13] in two main aspects, which we discuss next.

The first regards the initialization phase of the resilient algorithm. Note that if the adversary fails even a single node before the start of the computation, then this node's input is lost forever. The solution taken by [13] is promising $1/(1-\alpha)$ *quiet* rounds where no crashes can happen. These quiet rounds can be used to encode and distribute the nodes inputs. We take a different approach: we consider the inputs at the start of the computation as *encoded versions* of the inputs to the Congested Clique model (similar to Spielman's coded computation [45]). Since our robust algorithm is such that the outputs are also encoded in this manner, we get *composability* for free – we can simply start another computation after completing a former one.

Second, instead of using block error correction codes, we use *locally decodable codes (LDCs)* [31, 46], which allow a node to query only a small number of other nodes in order to decode only the information it needs for its next local computation. This yields a dramatic improvement in the round complexity of computing a circuit because of its huge effect on congestion. Working with LDCs gives rise to new challenges, because decoding an LDC codeword can be easily targeted by the adversary which can crash so many nodes. The crux of the proof of Theorem 2 shows how to overcome this, and that in fact it can be implemented in a *deterministic* manner.

Theorem 1 is immediately established by combining our robust computation of circuits in Theorem 2 with the natural conversion of any Congested Clique algorithm to a circuit (see a formal proof in, e.g., [13]).

▶ **Lemma 3.** *Let* ALG *be a Congested Clique algorithm that computes a function f in T rounds, where the node's input and randomness strings are of size $\widetilde{O}(n)$. Then, there is a circuit with depth $d = 2T+1$, width $\omega = \widetilde{\Theta}(Tn^2)$, and maximal gate total fan of $\Delta = \widetilde{O}(Tn)$, whose inputs are all the $\widetilde{O}(n^2)$ input bits of the nodes and whose outputs are the outputs of the nodes after running* ALG.

1.1 Technical Overview

Our main theorem states that we can robustly compute a circuit C of depth d and arbitrary gates, despite a possible (worst-case) crash of an α fraction of the nodes. Towards this end, the network computes the gates of C, layer by layer, where each node is assigned some of the gates in each layer in a dynamic manner that adapts to node failures.

As mentioned above, when a node crashes, it can no longer send messages, and thus any information that was privately held in that node's memory is lost forever. To be resilient to crashes and avoid losing important information, we need to store information "in the network" so that it can be retrieved despite a constant fraction of node crashes. Indeed, [13] used block error correction codes (ECCs) to encode data and split the resulting codewords across the network. This way, each piece of information can be retrieved by querying all the nodes for their part of the codeword; the decoding succeeds even if a constant fraction of the nodes crash, where the constant depends on the strength of the ECC in use. The drawback is that even if a single bit of information is needed from a coded string, its entire codeword needs to be decoded.

Our starting point is that we replace block codes with Locally Decodable Codes (LDCs). Informally, such codes allow decoding specific *parts* of the message, rather than decoding the entire message. Furthermore, decoding does not require obtaining the entire (corrupted) codeword, but rather queries relatively small parts of it (a subpolynomial number of symbols), while still guaranteeing correct decoding of the desired symbol with a high probability.

Hence, switching to LDCs benefits our algorithm in the sense that nodes make *less queries* in order to retrieve *exactly* the information they need for the computation. The advantage of LDCs over standard (i.e., block-codes) ECCs becomes clear when considering the case where some node is assigned to compute a gate with n inputs, each of which is stored in a different codeword. With standard ECCs, this means that the node must access all n codewords and retrieve all information bits, i.e., n^2 bits, assuming n-bit codewords, which causes high congestion.

Algorithm Overview. The high-level idea of our robust circuit computation algorithm is as follows. Consider a circuit C, whose inputs are distributed over the network in an encoded manner using some LDC code. The network computes C layer by layer. That is, let $\mathsf{gates}(1)$ be the first layer of gates in C, i.e., all the gates whose inputs are the inputs of C. We first distribute the tasks of computing these gates across the (non-crashed) nodes so that each node is assigned roughly the same number of gates to compute. Each node then attempts to compute all the gates allocated to it. To do this, the node first obtains the inputs for each gate assigned to it by decoding the corresponding inputs of C, which are stored in the network using codewords of an LDC. If this information retrieval is successful, the node computes the outputs of the gates allocated to it, and then "stores" them in the network by encoding them with an LDC and distributing the codewords to the nodes in the network. Once the first layer is computed and stored in the network, the nodes continue to compute the second layer of gates in C, denoted $\mathsf{gates}(2)$, which includes all gates whose inputs are either the inputs of C or the outputs of the gates in the first layer, $\mathsf{gates}(1)$. This continues until all the outputs of C are computed and stored in the network. Naturally, crashes that occur during the algorithm may prevent the network from completing the computation of a specific layer and progressing to the next layer, which we discuss next.

Overcoming Crashes I. In our robust circuit computation algorithm, each gate is assigned to a dedicated node responsible for its computation. If that node crashes, the gates assigned to it remain uncomputed. A trivial solution is to reassign any uncomputed gate in the current layer of C (whose original node is crashed) to a new node that is not crashed. However, the adversary could then crash this new node and eventually cause a delay of αn rounds, which is extremely expensive. Our strategy is different: when a node crashes, we reassign its gate(s) to *two* fresh nodes. If both of those nodes crash, we again double the redundancy, forcing the adversary to double its effort to keep the gate uncomputed. After at most $\log n$ such iterations, every gate is guaranteed to be computed by at least one live node. This progressive doubling remains feasible without causing excessive congestion because of two factors: First, the nodes that do not crash successfully compute and store their assigned gates. These nodes are now available to take over the gates of the crashed nodes. Second, the adversary cannot (effectively) corrupt too many nodes in the same round: If the adversary crashes too many nodes during a short period of time, we call this step *overwhelmingly faulty*, and simply restart the computation of this layer with the remaining nodes. While this translates to no progress, it reduces the adversary's budget of crashes and hence cannot occur too many times.

Deterministic LDCs and Congestion. The decoding algorithm of LDCs is *inherently random*. Indeed, if a fixed (small) number of codeword symbols are queried during a decoding attempt and these symbols are corrupted, then decoding is certainly impossible. If so, the code cannot correct a constant fraction of corruptions, as would be normally expected. In particular, given a budget of αn node crashes, an all-knowledgeable adversary may be able to crash a subset of nodes in a way that prevents any meaningful progress of the deterministic computation.

Despite the above conundrum, our robust algorithm is fully deterministic. In particular, we derandomize the LDC decodings performed throughout the computation while maintaining resilience to αn node crashes. Our derandomization relies on two important properties, specific to our model. First, when a node crashes, all other nodes are aware of this event because the crashed node does not send any messages from the round in which it crashed. This allows the remaining nodes to maintain a consistent view of the crashed and alive set of nodes. Second, crashed nodes that are queried for their respective parts of the LDC codeword do not reply, and thus the LDC decoding algorithm is missing some parts of the queried codeword, known as *erasure* corruptions. These are easier to correct than when the codeword contains incorrect information. The combination of erasure corruptions and knowledge of which nodes have crashed in each round allows a decoder to predict whether a specific set of queries will result in successful decoding. Thus the decoder can pick a set of queries that is guaranteed to succeed if no further nodes crash.

While the above idea derandomizes the (inherently random) LDC decoding algorithm, it creates a new challenge regarding the resulting congestion. To illustrate this challenge, suppose that a node performs the above deterministic selection of queries *separately* for each piece of information it wants to retrieve. Then, it may end up querying the same subset of nodes over and over again, thus causing a large congestion. Randomized decoding averts this issue by querying a set of nodes in a near-uniform distribution. However, even if we could deterministically replicate this querying distribution, we face again the issue mentioned above, where many of these queries are erased and do not lead to a correct decoding.

Nevertheless, our analysis, which is based on the probabilistic method, shows that it is possible to select query sets for multiple (independent) LDC-decoding instances in the presence of a constant fraction of erasures in positions known to the decoding algorithm, so that the following hold simultaneously: (i) each decoding instance successfully decodes the correct information, and (ii) the congestion per queried node is small, i.e., the queries are well-distributed over the network. To show the latter, we analyze an equivalent bins-into-balls experiment, showing that the event that too many balls (queries) aggregate in one specific bin (node) happens with small probability. Union-bounding over all nodes keeps the probability of the bad event below 1, thus proving the existence of good query sets that avoid congestion.

Overcoming Crashes II. The above discussion implies that the adversary *cannot* select a set of αn nodes to crash for invalidating many of the LDC decoding attempts throughout the computation, as long as these indices are known to the nodes. However, the adversary may decide to crash nodes *after* a decoder fixes its selection of nodes to query in a given round, as this selection depends only on nodes that are crashed *prior* to that round. To overcome this problem, the nodes dynamically increase the number of times they attempt to LDC decode each piece of information, according to corruptions made so far. Namely, if the decoding of some LDC-encoded information fails due to *new* corruptions of the queried nodes, then the decoding node performs *two* independent decoding attempts. These new queries depend on all the crashes so far, and in particular, on the "new" crashes that invalidated the original decoding attempt. If these two attempts fail as well (due to new crashes that occur after the nodes queried by these two attempts are decided), the decoding node doubles its number of attempts again, and so on. Overall, after a logarithmic number of doublings, this approach potentially causes a large, near-linear number of LDC decoding attempts, and the adversary can only fail a constant fraction of them without exceeding its budget. Note that it only takes one successful attempt to move on, so the adversary must fail all attempts of a single codeword to prevent progress.

Recap. We can now summarize the overview of our robust circuit computation. For a given circuit C, the computation goes layer by layer, where computing a layer of C means: (1) assigning uncomputed gates of the layer to non-crashed nodes; (2) retrieving the inputs to the gates of this layer, that are stored in the network via an LDC during the computation of previous layers; (3) computing the gates; (4) storing the outputs of the gates via an LDC. Technically speaking, this computation of each layer is done in two nested loops: The external loop doubles, in each iteration, the number of nodes responsible for computing some uncomputed gate (we call this loop *the* NodeDoubling *loop*). The internal loop doubles, in each iteration, the number of independent decoding attempts each node makes for each uncomputed gate assigned to it (we call this loop *the* AttemptDoubling *loop*). Section 3 fully details our algorithm. The algorithm's analysis and the derandomization of the LDC in our setup appear in the full version of the paper [8].

1.2 Additional Related Work

Since its introduction for faster MST computation [36], the Congested Clique model has been extensively explored during recent decades for various tasks. The MST complexity was eventually shown to be constant [40] following a beautiful line of work [26, 28, 30, 32]. Additional examples include routing [35], coloring [4, 14, 16, 17, 42, 43], subgraph finding [7, 9, 10, 19, 21, 29, 41], and many more. Hardness of obtaining lower bounds in this model is established in [20].

Fault-tolerance in the Congested Clique model was explored by [33,34] for graph realization problems under crash-faults, by [3, 15] for recognizing connectivity and hereditary properties under Byzantine faults, and by [23, 24] for general computations under edge faults. In particular, [24] employs LDCs as means to concentrate information from many nodes into few.

Coding theory, in various forms, has been extensively used in many other areas of distributed computing, including: distributed zero-knowledge [5, 27], proof labeling schemes [11, 22], the beeping model [1, 18, 25], and distributed interactive proofs [39].

2 Preliminaries

For an integer $n \geq 1$ we denote $[n] = \{1, 2, \ldots, n\}$. All logarithms are taken to base 2 unless otherwise mentioned. We say that an event occurs with high probability (in n, which is usually implicit) if its probability is at least $1 - 1/n^{10}$. For a string x and for any $i \in [|x|]$, let $x[i]$ denote the i-th symbol of x.

2.1 Computation Model

Suppose a Congested Clique network, where n nodes, $v_1, \ldots, v_n$, communicate in synchronous rounds by exchanging $b \log n$-bit messages in an all-to-all fashion, for some constant $b \in \mathbb{N}$. Throughout the computation, an adversary may choose to crash up to αn nodes, where the constant $\alpha \in [0, 1)$ is a parameter of the model. A crashed node does not send any messages starting from the round in which it is crashed. The corruption is *worst case*: an all-knowledgeable adversary bases its decision on which nodes to fail on all its available information, including the algorithm that the nodes execute, their inputs, and their local randomness (if any). Note that all nodes know which nodes are non-faulty at the end of each round, denoted as the set $\mathcal{A}$ (to indicate that they are *alive*).

The compiler of Algorithm 5 is completely deterministic, in the sense that it does not add any new randomness. In particular, The robust algorithm ALG′ remains deterministic if ALG was deterministic, and similarly, it is randomized if ALG was so. In the latter case, we assume that the randomness of ALG is given to the representing circuit C as input.

Coded inputs and outputs. In the Congested Clique model, it is common to refer to problems in which each node holds a private input x of $O(n \log n)$ bits. As explained in the introduction, in our faulty model, the inputs must be coded in a way that prevents them from being lost if some node crashes before the first round of communication. We thus consider inputs as encoded via an LDC code (see Definition 4 in Section 2.3 below), and the respective codeword is distributed across the nodes of the network. We employ a (q, δ, ϵ)-LDC code with block length n, whose parameters will be specified later. Such a code guarantees that every bit of the input x (of each node) can be retrieved by querying q nodes with probability $1 - \epsilon$ over a uniform choice of the randomness string, even if δn of the nodes have crashed. It will be the case that $\alpha < \delta$.

We require the output to be stored in the network in a similar way: the outputs should be encoded via an LDC code whose resulting codewords are split among the nodes, so that it is possible to retrieve each bit of the output despite δn crashes. This choice allows the composition of computations, where the outputs of one computation are the inputs of the next.

2.2 Layered Circuits

We identify a circuit C with a directed acyclic graph $C = (V_C, E_C)$ in which every gate is associated with a node, and every wire connecting gates is associated with an edge. Each bit-input to the circuit (an input gate) is associated with a leaf node and each output of the circuit (an output gate) with a root node. Other nodes are associated with the computational gates of the circuit. We say that a node $g \in V_C$ *depends* on a node $g' \in V_C$ if there is a directed path from g' to g. The notion of gate dependencies induces *layers* in the circuit, where all input gates are in layer 0, and a gate g is placed in layer i if i is the minimal integer such that g depends only on nodes in layers at most $i - 1$. For a gate $g \in V_C$, we denote by $\mathsf{layer}(g)$ the layer of g, and let $\mathsf{gates}(i) = \{g \in V_C \mid \mathsf{layer}(g) = i\}$ be the set of all the gates in layer i. The depth of C, denoted $d = d(C)$, is defined to be $\max_{g \in V_C} \mathsf{layer}(g)$. We denote by $\mathsf{wires}(i) = \{(u, v) \in E_C \mid \mathsf{layer}(u) = i\}$ the set of all wires that go out of the gates in layer i. Note that $\mathsf{wires}(0)$ are the inputs to the circuit. For a gate g, denote by $\mathsf{fan\text{-}in}(g)$ its in-degree and by $\mathsf{fan\text{-}out}(g)$ its out-degree; let $\mathsf{fan}(g) = \mathsf{fan\text{-}in}(g) + \mathsf{fan\text{-}out}(g)$ be its *total fan*. The *width* of the circuit, $\omega = \omega(C)$, is defined as the maximal number of outgoing wires of any layer, $\omega = \max_i(\mathsf{wires}(i))$. We assume throughout that all parameters of the circuit are polynomial in the size of the network, i.e. $d, \omega, \Delta = O(\mathrm{poly}(n))$. This fits the case where C represents a Congested Clique algorithm with $O(n \log n)$ bits of input per node. We note, however, that the statement of Theorem 2 holds, up to logarithmic terms, for any parameters d, ω, and Δ.

2.3 Error Correcting Codes and Locally Decodable Codes

For an alphabet Σ, the Hamming distance of two strings $x, y \in (\Sigma \cup \bot)^*$ of the same length, i.e., $|x| = |y|$, is the number of indices for which x and y differ and is denoted by $\mathrm{Hamm}(x, y) = |\{i \mid x[i] \neq y[i]\}|$. For two strings $x \in \Sigma^*$, $y \in (\Sigma \cup \{\bot\})^*$ and value $c \in (0, 1)$, we say that y can be obtained by a c-fraction of erasures from x if $|x| = |y|$, and for all $i \in [|x|]$, it holds that either $x[i] = y[i]$ or $y[i] = \bot$, where the latter case happens at most $c|x|$ times. An index i in which $y[i] = \bot$ is called an erasure.

For a prime power p, we denote by $\mathbb{F}_p$ the finite field of size p. An *error correcting code* is a mapping $\mathsf{Enc} : \mathbb{F}_p^K \to \mathbb{F}_p^N$ that takes K symbols of the alphabet $\mathbb{F}_p$ into N symbols of the alphabet $\mathbb{F}_p$.[1] The value N is called the block length of the code. The ratio K/N is called

[1] We can map $[p]$ and $\mathbb{F}_p$ with a fixed isomorphism, so that $\mathsf{Enc} : [p]^K \to [p]^N$.

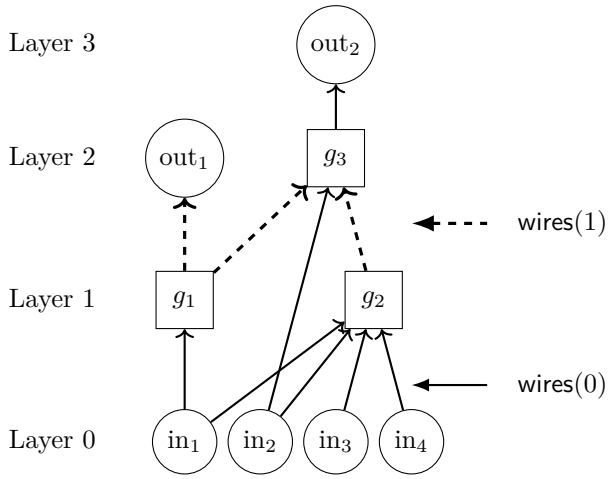

Figure 1 An example of a circuit C of depth $d = 3$ and width $\omega = 6$. We have $\mathsf{gates}(1) = \{g_1, g_2\}$ and $\mathsf{gates}(2) = \{g_3\}$. The gate g_2 has $\mathsf{fan\text{-}in} = 4$ and $\mathsf{fan\text{-}out} = 1$ giving $\mathsf{fan} = 5$, while the gate in_1 has $\mathsf{fan\text{-}out} = 2$ and $\mathsf{fan} = 2$. The set $\mathsf{wires}(0)$ and $\mathsf{wires}(1)$ are indicated on the figure.

the *rate* of the code. The *relative distance* of a code is the normalized Hamming distance between any two codewords, denoted $\delta = \min_{m \neq m'} \frac{1}{N}\mathrm{Hamm}(\mathsf{Enc}(m), \mathsf{Enc}(m'))$.

Next, we formally define the notion of locally decodable codes. For the purposes of our work, we only consider the *erasure* setting, in which we are given access to a possibly corrupted codeword y, obtained by erasing at most δ-fraction of some $\mathsf{Enc}(x)$ for some $x \in \mathbb{F}_p^K$, and an index $i \in [K]$, and our goal is to find the i-th symbol of x.

▶ **Definition 4** (Locally Decodable Codes (LDCs) for erasures). *An error correcting code* $\mathsf{Enc} : \mathbb{F}_p^K \to \mathbb{F}_p^N$ *is said to be a* (q, δ, ϵ)*-LDC if there exists a randomized decoding algorithm* Dec *that receives as input a string* $y \in (\mathbb{F}_p \cup \{\perp\})^N$ *and an index* $i \in [K]$*, performs at most* q *queries to* y*, and outputs a value with the following guarantee: if there exists* $x \in \mathbb{F}_p^K$ *such that* y *can be obtained by at most a* δ*-fraction of erasures from* $\mathsf{Enc}(x)$*, then* $\Pr(\mathsf{Dec}(y, i) = x[i]) \geq 1 - \epsilon$.

▶ **Definition 5** (Non-adaptiveness). *An LDC is called* non-adaptive *if for every call to its decoding algorithm* Dec*, the set of queries it performs given input index* i *is only a function of the randomness and the index* i*. In particular, a query does not depend on the outcome of previous queries.*

We can think of the decoding algorithm of a non-adaptive LDC code $\mathsf{Dec}(\cdot, i)$ as an algorithm with oracle access to the codeword, that first generates q indices to query, and then, once provided these (possibly corrupt) q symbols, returns the decoded message symbol.

The following smoothness property of LDCs means that decoding an index requires querying the codeword in a "smooth" (near-uniform) way. This property is important in order to avoid congestion when a node decodes multiple values.

▶ **Definition 6** (Smoothness). *An LDC is called* s-smooth *if there exists a decoding algorithm* Dec*, such that during any call to* Dec*, any entry* $j \in [N]$ *of the codeword is queried with probability at most* s*.*

The following theorem suggests that smoothness is an inherent property of LDCs, since any decoding algorithm can be transformed into a smooth one.

▶ **Theorem 7** ([31, Theorem 1]). *Every* (q, δ, ϵ)*-LDC of block length* N *is* $q/\delta N$*-smooth.*

3 Computing a Circuit in the presence of crashes

We show how to efficiently and deterministically compute a specified circuit C in the Congested Clique model, in the presence of up to αn crashes. We start, in Section 3.1, by describing the procedures Store, Retrieve, and BulkRetrieve used to store and retrieve information in our algorithm. In section Section 3.2 we describe another procedure, Allocate, that assigns gates to nodes in a balanced-manner. Finally, in Section 3.3, we describe our circuit computation algorithm based on these procedures.

3.1 The Store, Retrieve, and BulkRetrieve Procedures

As mentioned above, any information that may get lost due to node crash is stored in the network via an LDC. We first describe the LDC we use and then detail the store and retrieve procedures.

The LDC instantiation

We assume a fixed LDC code, whose exact details appear in the full version of the paper [8]. Specifically, for some power of prime $q = 2^{O(\sqrt{\log n})}$ and ρ such that $\rho^{-1} = 2^{O(\sqrt{\log n}\log\log n)}$, our LDC has an encoding function $\mathsf{Enc} : [q]^{\rho n} \to [q]^n$ and a decoding function Dec which, given an input index $i \in [n]$, smoothly queries q indices of the codeword and decodes correctly even if up to δ-fraction of the q queried symbols are erased, for some predetermined constant distance $\alpha < \delta < 1$. We assume that $n = q^r$ for some integer r (specified in the detailed construction); this assumption can be lifted using standard methods. Note that Definition 4 says that Dec is randomized, but our goal is to compute the circuit deterministically. We thus treat any randomness used by Dec as originating from some *randomness string*, but our implementation of obtaining such randomness strings will be *deterministic* rather than randomized which will render our implementation of Dec, and hence our circuit computation algorithm, deterministic.

The Store Procedure

The Store procedure "saves" information in the network in a robust way, by encoding it with an LDC and distributing the codeword among the nodes of the network.

Each node v_j begins the Store procedure with a bit-string U_j it wishes to store in the network. It first splits U_j into parts of size $\rho n \lfloor \log q \rfloor$ bits each (so that each can be represented by a string of ρn symbols over the alphabet $[q]$), padding with zeros as necessary. Set $\mathsf{last}_j = \lceil |U_j| / (\rho n \lfloor \log q \rfloor) \rceil$ and denote these parts $U_j^1, \ldots, U_j^{\mathsf{last}_j}$. The node v_j then encodes each U_j^i using Enc to obtain a codeword $\mathsf{Enc}(U_j^i) = L_j^i$ of size $|L_j^i| = n$ symbols.

Next, v_j distributes the codewords to the network nodes. Specifically, in round $i = 1, \ldots, \mathsf{last}_j$, it sends the symbols of L_j^i – one symbol to each node in the network. This takes a single round of communication because each symbol in the LDC codeword comes from the alphabet $[q]$ with $q = 2^{O(\sqrt{\log n})}$, hence it can be encoded in $\log q = O(\log n)$ bits. The formal description is depicted in Algorithm 1.

We say that v_j *stored* U_j in the network if v_j completed the Store procedure without crashing. The following is straightforward from Algorithm 1.

▶ **Observation 8.** *Storing a string U_j takes $O(\lceil |U_j| / (\rho n \lfloor \log q \rfloor) \rceil)$ rounds of communication.*

Algorithm 1 Store (for node v_j).

Input: A bit-string U_j.

1: Partition U_j into consecutive parts of size $\rho n \lfloor \log q \rfloor$ bits, padding the last part if necessary; denote these substrings $U_j^1, \ldots, U_j^{\mathsf{last}_j}$. ▷ $\mathsf{last}_j = \lceil |U_j| / (\rho n \lfloor \log q \rfloor) \rceil$
2: **for** $i = 1, \ldots, \mathsf{last}_j$ **do**
3: $L_j^i \leftarrow \mathsf{Enc}(U_i^j)$.
4: For all $t \in [n]$ in parallel, send $L_j^i[t]$ to v_t.
5: **end for**

The Retrieve and BulkRetrieve Procedures

In the Retrieve procedure, a node v_j is given as input an index w of some string U that was previously stored in the network using the Store procedure. Node v_j is additionally given a string R called the randomness string. One can think of this procedure as randomized, with R as its randomness, but in all our invocations of Retrieve, the string R is set deterministically, as will be explained later.

The goal of the Retrieve procedure is to retrieve the value of the w-th bit of the previously stored U. To that end, v_j first identifies the codeword that contains the bit w: recall that the Store procedure splits U into parts of size $\approx \rho n \log q$ bits. Denote by U^i the respective part and by i' the index of the symbol in U^i that contains the bit-value $U[w]$ in which we are interested. In the following, we say "decode $U[w]$" to actually mean decoding the respective index i' of the possibly corrupted codeword $\mathsf{Enc}(U^i)$ that contains the respective value.

To retrieve the value of $U[w]$ from the (stored) codeword $\mathsf{Enc}(U^i)$, the node v_j executes $\mathsf{Dec}(.)$ using the randomness string R and obtains indices of random q symbols of $\mathsf{Enc}(U^i)$ needed for the decoding. It is possible to learn these q indices in advance because the LDC is non-adaptive (Definition 5). The node v_j then queries the respective nodes for their stored symbols and provides $\mathsf{Dec}(.)$ with their replies. Note that crashed nodes do not reply, which translates to erasures given to $\mathsf{Dec}(.)$. Further, in hindsight, the randomness string R will be derandomized, which has the effect that all nodes know R. It will follow that v_j does not actually need to send any message in order to query any node; the q nodes will know that they are the nodes that should give information back to v_j, since they will know the identity of i', U^i and the value of R to begin with. See Lemma 14.

Under some circumstances, we allow the retrieval to fail, in which case the output is $\perp$. This could happen in two cases: (i) when there are too many erasures and $\mathsf{Dec}(.)$ returns $\perp$, or (ii) if one of the nodes queried during this Retrieve invocation crashes during the execution of this Retrieve invocation. For the former, our derandomization will guarantee that this event cannot happen if at most αn nodes have crashed. For the latter, in this case we set the output to be $\perp$ even if the decoding Dec successfully retrieves the symbol. This decision does not affect the correctness of the algorithm, but rather simplifies its analysis.

The BulkRetrieve procedure generalizes the above to allow retrieving multiple previously stored bits. Now, v_j is given as input a *collection* of indices W_j, where each index $w \in W_j$ refers to some (predetermined) $U_{(w)}$ that was previously stored in the network via a Store. The strings $U_{(w)}$ may be different for different values of w, or they may be the same. Additionally, the procedure gets a multiplicity parameter ℓ. The goal is to output the value of the bit in index w of $U_{(w)}$, for each index $w \in W_j$.

Algorithm 2 Retrieve (for node v_j).

Inputs: An index w to some string U previously stored in the network via the Store procedure and a randomness string R.

1: Identify the part U^i used by Store to encode $U[w]$ and the respective index i' in it that contains that value, i.e., the symbol $U^i[i']$ contains the bit $U[w]$.
2: Execute $\mathsf{Dec}(\cdot, i')$ with randomness string R, to obtain the q indices needed to decode $U^i[i']$. Set $S \subset V$ to be the nodes that hold these respective symbols.
3: Query the respective nodes in S. Treat symbols associated with crashed nodes as erasures.

Output:
4: If some node $v \in S$ crashes **during the execution of Line 3**, output $\perp$.
5: Otherwise, output the bit $U[w]$ contained in $\mathsf{Dec}(\mathsf{Enc}(U^i, i'))$ by providing the q replies (including erasures) when Dec queries the codeword $\mathsf{Enc}(U^i)$.

Towards this goal, all nodes in the network first *deterministically* compute a set of randomness strings $\mathcal{R}(v_j) = \{R_{v_j,w,i} \mid i \in [2^\ell], w \in W_j\}$ for each $v_j \in V$, with *good* properties, which we define and discuss in detail later in the section (see Definition 9). The deterministic generation of these strings is given by Lemma 10. After this step, all nodes $v \in V$ know $\mathcal{R}(v_j)$ for every v_j.

Next, v_j performs, for each index $w \in W_j$, a batch of 2^ℓ Retrieve procedures, where the i-th invocation uses randomness string $R_{v_j,w,i}$. Similar to the case of Retrieve, we allow some retrieves to fail and output $\perp$. If at least one of the 2^ℓ invocations of $\mathsf{Retrieve}(w, R_{v_j,w,i})$ succeeds, its output becomes the output of BulkRetrieve for the index w; otherwise, the respective output is $\perp$.

All the $|W_j| \cdot 2^\ell$ Retrieve invocations are executed in parallel. However, in order to avoid congestion, v_j pipelines requests targeted to the same node. That is, it sends at most one query to any node in any given round. Similar to above, the set of strings $\{U_{(w)}\}_{w \in W_j}$ is predetermined and known to all nodes, and the identities of the q nodes that are queried in a specific $\mathsf{Retrieve}(w, R_{v_j,w,i})$ are generated using $R_{v_j,w,i}$, which is also known to all nodes. Hence, each queried node can infer the respective $U_{(w)}$ for each query, without the need for v_j to communicate this data. The formal procedure is depicted in Algorithm 3.

Consider a specific instance of $\mathsf{BulkRetrieve}(W_j, \ell)$. The selection of randomness strings $\mathcal{R}(v_j)$ that v_j uses has a tremendous effect on the induced congestion. Indeed, assume that $\mathcal{R}(v_j)$ is such that *all* Retrieves query some v_i, implying $2^\ell |W_j|$ rounds of communication where in each round a single LDC symbol is communicated. This should be contrasted with the fully randomized case, where each node is queried in a near-uniform distribution (implied by the smoothness of LDC codes, Theorem 7), implying that each v_i is queried $2^\ell |W_j| \cdot q/n$ times, in expectation. Standard tail bounds show that the number of queries of the maximal node (and hence the round complexity) is bounded by $2^\ell |W_j| \cdot q/n \cdot O(\log n)$. This gives that there *exists* a way to select the randomness strings $\mathcal{R}(v_j)$ while maintaining the same round complexity.

However, while the above gives uniform query locations for the goal of controlling congestion, it does not address the problem that many locations might be erased. To illustrate this point, assume that out of the 2^ℓ Retrieve instances, all but one are querying mostly erased symbols (crashed nodes), and only one Retrieve correctly decodes the value. The output of BulkRetrieve would be correct in this case, but in this scenario the adversary needs to crash only a single additional node in order to fail it.

Algorithm 3 BulkRetrieve (for node v_j).

Inputs: An ordered collection W_j of indices, where each $w \in W_j$ refers to a predetermined string $U_{(w)}$, that was previously stored by some node executing Store. A multiplicity parameter ℓ.

1: Compute a collection of randomness strings $\mathcal{R}(v_j) = \{R_{v_j,w,i} \mid w \in W_j, i \in [2^\ell]\}$ using Lemma 10.
2: **parallel for** each $w \in W_j$ **do**
3: **parallel for** $i = 1, \ldots, 2^\ell$ times **do**
4: Run Retrieve$(w, R_{v_j,w,i})$. If v_j needs to query the same node multiple times in all of these parallel instances, send at most one query per round, until all queries are sent.
5: **end parallel for**
6: **end parallel for**

7: **Output:** For each w, the output is the output of the first successful Retrieve$(w, R_{v_j,w,i})$, if any, or $\perp$ otherwise.

Indeed, what we show is even stronger than mimicking a fully randomized case by some naïve load balancing. The following Lemma 10 shows that we can find a set of randomness strings that maintains a similar round complexity even if each codeword has αn symbols erased, and furthermore, *each individual* Retrieve *succeeds decoding the respective value*, as long as no new crashes happened during that Retrieve. In other words, we can de-randomize the random sampling of codeword symbols to query while (i) maintaining complexity (by controlling congestion) and (ii) performing only "useful" queries, hence maintaining our resilience to an all-knowledgeable adversary. Our choice of randomness strings guarantees that the adversary must waste 2^ℓ of its crashing budget in order to fail the BulkRetrieve, which is crucial for the correctness proof.

We now define the notion of good randomness strings, namely, strings that provide the above properties for the BulkRetrieve procedure.

▶ **Definition 9** (Good randomness strings). *Fix a node v_j, parameters W_j, ℓ, and the set of non-crashed nodes $\mathcal{A}$. A collection of randomness strings $\mathcal{R}(v_j) = \{R_{v_j,w,i}\}_{w \in W_j, i \in [2^\ell]}$ is called* good *for an instance of* BulkRetrieve(W_j, ℓ)*, if the following holds:*

(1) *Each node is queried at most $O(\lceil \frac{2^\ell |W_j| q}{n} \rceil \log n)$ times in total by v_j, and*

(2) *For all $w \in W_j$ and $i \in [2^\ell]$, the invocation of* Retrieve$(w, R_{v_j,w,i})$ *succeeds (given no further changes in $\mathcal{A}$).*

In the full version of the paper [8], we prove the following.

▶ **Lemma 10.** *There is a zero-round deterministic algorithm which, given the set of non-crashed nodes $\mathcal{A}$, a collection of indices W_j, and a multiplicity parameter ℓ, computes a good collection of randomness strings $\mathcal{R}(v_j)$ for v_j.*

With the above, the following is immediate.

▶ **Lemma 11.** BulkRetrieve(W_j, ℓ) *takes $O(\lceil \frac{2^\ell |W_j| q}{n} \rceil \log n)$ rounds.*

3.2 The Allocate Procedure

The purpose of the Allocate procedure is to assign a set of given gates that need to be computed to non-crashed nodes. The procedure is deterministic and runs locally on each node, without any communication. However, all nodes reach the same allocation, since they all have the same knowledge regarding crashed nodes and regarding failed LDC queries, where the latter is due to the randomness strings being generated deterministically by all nodes and known to all (due to Lemma 10 above and the upcoming Lemma 14 which essentially says that the nodes are able to keep a consistent view of all of these variables by careful bookkeeping).

In more detail, the procedure is given as input the current set of non-crashed nodes $\mathcal{A}$, a set of gates $G \subseteq V_C$ (of the circuit $C = (V_C, E_C)$, known to all), and a multiplicity parameter ℓ_1. For each gate $g \in G$, Allocate assigns g to a set of $\min(2^{\ell_1}, |\mathcal{A}|)$ nodes from $\mathcal{A}$ using the following sequential "greedy" process: Sort the gates in G by their total fan (denoted fan), in descending order. Assign the gates one by one to a set of $\min(2^{\ell_1}, |\mathcal{A}|)$ distinct nodes in $\mathcal{A}$ whose loads are minimal (break symmetry by node IDs). The *load* of a node v, denoted $\lambda(v)$, is defined as the sum of the total fan of all gates assigned to it so far during this Allocate instance. See Algorithm 4.

Algorithm 4 Allocate (for node v_j).

Inputs: A set G of gates and a multiplicity parameter ℓ_1.

Let $g_1, \ldots, g_{|G|}$ be the gates of G, sorted in descending order of fan (break ties consistently).
Set $\lambda(v) \leftarrow 0$ for all nodes $v \in \mathcal{A}$.
$G_j \leftarrow \emptyset$.
$L \leftarrow \min(2^{\ell_1}, |\mathcal{A}|)$.
for $i = 1, \ldots, |G|$ **do**
    Let $u_1, \ldots, u_L$ be the L nodes in $\mathcal{A}$ with the minimal loads (break ties by IDs).
    for each node $u \in \{u_1, \ldots, u_L\}$ **do**
        Assign g_i to the node u.
        $\lambda(u) \leftarrow \lambda(u) + \mathsf{fan}(g_i)$.
        if $u = v_j$ **then** $G_j \leftarrow G_j \cup \{g_i\}$.
        end if
    end for
end for

Output: The set G_j of gates assigned to v_j.

The assignment can be computed locally in a consistent manner across all non-crashed nodes without any communication, since all relevant information, namely G, ℓ_1, and $\mathcal{A}$, is known to all nodes in $\mathcal{A}$. Note that since this is a local computation procedure, we can assume that set $\mathcal{A}$ does not change throughout the computation, and that all nodes use the same set $\mathcal{A}$ representing the non-crashed nodes at the beginning of that round.

The following lemma bounds the load assigned to each node, and is proven in Section B.

▶ **Lemma 12.** *Let* $L = \min(2^{\ell_1}, |\mathcal{A}|)$, $P = \sum_{g \in G} \mathsf{fan}(g)$, *and assume* $\max_{g \in G} \mathsf{fan}(g) \leq \Delta$. *Then,* Allocate$(G, \ell_1)$ *puts a maximal load of* $\max(4PL/|\mathcal{A}|, \Delta)$ *on each node.*

3.3 The Circuit Computation Algorithm

We can now complete the description of our circuit computation algorithm, presented in Algorithm 5. The algorithm takes as input a circuit C whose inputs (the wires $\mathsf{wires}(0)$) are already stored in the network.

The algorithm computes the gates of C layer by layer, in a sequence of d steps referred to as layer-*steps*. For layer-step $i = 1, \ldots, d$, we assume that $\mathsf{wires}(0), \ldots, \mathsf{wires}(i-1)$ have already been stored by previous iterations, and the goal is to compute and store $\mathsf{wires}(i)$. To this end, the nodes execute Allocate, which assigns to each non-crashed node v_j a set of gates $G_j \subseteq \mathsf{gates}(i)$ to compute and store their output wires (line 5).

Then, each node v_j tries to retrieve the input wires of the gates G_j assigned to it via the BulkRetrieve procedure (line 8). If successful, the node computes the gates assigned to it (line 12) and obtains the values of all output wires U_j of the gates G_j. The node then stores these wires in the network (line 13).

However, crashes that occur during this computation may hinder the computation of some wires. To overcome this issue, the computation of layer i consists of two nested loops. The outer loop, which we call the NodeDoubling-loop, iterates over $\ell_1 = 1, \ldots, \lceil \log n \rceil$ and doubles the number of nodes that try to compute a given gate. The inner loop, called the AttemptDoubling-loop, iterates over $\ell_2 = 1, \ldots, \lceil \log \Lambda \rceil$ for some parameter Λ (fixed in the analysis), and doubles the number of retrieval attempts a given node performs for each input wire assigned to it.

If during the computation of layer i, more than $c_f n/(q \log n)$ new crashes have occurred, for some sufficiently small constant $c_f > 0$ determined later, we re-start the computation of that layer with the remaining nodes. Namely, we maintain a counter $f_{i,rep}$ of newly crashed nodes in the layer-step, which is initialized at the start of the layer-step to be 0. Once it passes $c_f n/(q \log n)$, we reset the counter to 0, reset the multiplicity parameters ℓ_1, ℓ_2 to 1, and retry to compute and store all remaining unstored wires in $\mathsf{wires}(i)$. This action is captured in lines 16–20, assisted by the variable *rep*, that counts the number of repetition attempts of computing layer i. We call such a repetition of a layer-step *overwhelmingly faulty*:

▶ **Definition 13.** *Let $c_f > 0$ be a sufficiently small constant. A repetition of a* layer*-step is called overwhelmingly faulty if $c_f n/(q \log n)$ new crashes occur during this step.*

The next lemma captures the following observation: the nodes are capable of "bookkeeping" the progress of the computation at any given round of Algorithm 5. This bookkeeping information includes gates that were computed and stored, gates that still need to be computed, Retrieve calls that succeeded and those that failed, etc. In particular, when a node needs to access some wire w, that was previously stored, the node knows exactly which LDC codeword contains it, and which index of that codeword it should decode in order to retrieve w.

▶ **Lemma 14** (Bookkeeping)**.** *Any non-crashed node knows, at the start of any round, the following information: (1) the set S of gates whose outputs were stored in the network, and (2) for any $g \in S$ and any output w of g, the LDC codeword that contains w (namely, the node v_j that stored it and the round in which it was stored) and the index of w in the string U_j that v_j stored.*

Proof. Recall that, by definition, since a crashed node does not send any messages starting from the round in which it crashes, all nodes know the set $\mathcal{A}$ of non-crashed nodes at the beginning of every round.

Algorithm 5 Robust Circuit Computation (for node v_j).

Inputs:
A globally known circuit C.
The inputs wires(0) to the circuit C are stored in the network via the Store procedure.
Global parameters Λ, maxRetTime and maxStoreTime (to be set later).

1: **for** Layer $i = 1, \ldots, d$ **do** ▷ The layer-loop
2: $rep \leftarrow 1$, $S \leftarrow \emptyset$
3: **for** $\ell_1 = 1, \ldots, \lceil \log n \rceil$ **do** ▷ The NodeDoubling-loop
4: $G \leftarrow \mathsf{gates}(i) \setminus S$
5: $G_j \leftarrow \mathsf{Allocate}(G, \ell_1)$
6: Let W_j be the set of wires required for computing all gates in G_j.
7: **for** $\ell_2 = 1, \ldots, \lceil \log \Lambda \rceil$ **do** ▷ The AttemptDoubling-loop
8: Execute BulkRetrieve(W_j, ℓ_2). ▷ Idle until maxRetTime rounds pass
9: Remove from W_j all wires that were successfully retrieved in line 8.
10: **if** $W_j = \emptyset$ **then** ▷ Otherwise, idle
11: Let U_j be the output wires of G_j.
12: Locally compute the values of U_j using the retrieved values.
13: Execute Store(U_j). ▷ Idle until maxStoreTime rounds pass
14: **end if**
15: Update S to include gates whose output wires were stored by at least one node.
▷ If too many failures have occurred, repeat layer i
16: Let $f_{i,rep}$ be the number of crashes since the last execution of line 2 or 19.
17: **if** $f_{i,rep} > c_f n/(q \log n)$ **then**
18: $rep \leftarrow rep + 1$
19: **continue from** line 3, re-setting $\ell_1 \leftarrow 1$.
20: **end if**
21: **end for**
22: **end for**
23: **end for**

We prove Items (1) and (2) by induction on the round number. At initialization (the beginning of the first round, $r = 1$), the inputs to the circuit C are assumed to be stored, hence (1) and (2) hold for the input gates gates(0).

Next, we assume the statement holds at the beginning of some NodeDoubling-step, and we show it holds in every round until the end of this NodeDoubling-step. Note that all nodes execute Algorithm 5 in synchrony and, specifically, they all perform BulkRetrieve or Store at the same rounds (other actions do not involve communication as they are purely computational and thus take zero rounds).

If round r is not the final round of a Store procedure, then the set of stored wires is unchanged. Otherwise, each node knows the set S of stored gates at the beginning of the Store, by the induction hypothesis, and thus it also knows the set $G = \mathsf{gates}(i) \setminus S$. Since Allocate is deterministic and depends only on C, ℓ_1, G, and S, then all nodes learn the same output of $\mathsf{Allocate}(G, \ell_1)$. In particular, they all learn the gates G_j that each $v_j \in \mathcal{A}$ is assigned to compute in this NodeDoubling-step.

With this knowledge, all nodes can (locally) generate the good randomness strings that are used by some v_j for each of its BulkRetrieve invocations (Lemma 10). Further, all nodes have the same knowledge about Retrieve calls that failed in previous rounds due to new

crashes. They can thus infer which nodes v_j have successfully retrieved all input wires of G_j (i.e., those for which $W_j = \emptyset$) and satisfy the condition of in Algorithm 5. Only these nodes perform the Store that completes in that round r.

Out of the nodes that perform Store, any node v_j that does not crash before round r, succeeds in storing all the wires in U_j (i.e., all the output wires of G_j).

Therefore, at the start of round $r + 1$, all the nodes in $\mathcal{A}$ learn the set of nodes that performed a successful Store. They also know the set U_j of each v_j that completed a Store, in particular, which wires it contains and their internal order.[2] This implies Item (2). It also follows that all nodes in $\mathcal{A}$ can update their set S of stored gates in a consistent manner (line 15), which implies Item (1). ◀

As a result of this careful bookkeeping, v_j does not need to send any "metadata" information to the nodes it needs to query – they already have all the needed information (in the notations of BulkRetrieve, they know w and $U_{(w)}$, the randomness strings $\mathcal{R}(v_j)$, and the specific round(s) in which v_j queries them (i.e., is expecting a symbol from them).

This concludes the description of the algorithm. The algorithm's analysis, the description of the LDC we construct for the purpose of deterministic local decoding, and the proof of Lemma 10 are deferred to the full version of the paper [8].

References

1 Yagel Ashkenazi, Ran Gelles, and Amir Leshem. Noisy beeping networks. *Information and Computation*, 289:104925, 2022. doi:10.1016/j.ic.2022.104925.

2 Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

3 John Augustine, Anisur Rahaman Molla, Gopal Pandurangan, and Yadu Vasudev. Byzantine connectivity testing in the congested clique. In *36th International Symposium on Distributed Computing (DISC)*, volume 246, pages 7:1–7:21, 2022. doi:10.4230/LIPIcs.DISC.2022.7.

4 Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Efficient deterministic distributed coloring with small bandwidth. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 243–252, 2020. doi:10.1145/3382734.3404504.

5 Aviv Bick, Gillat Kol, and Rotem Oshman. Distributed zero-knowledge proofs over networks. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2426–2458, 2022. doi:10.1137/1.9781611977073.97.

6 Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. Fast approximate shortest paths in the congested clique. *Distributed Computing*, 34(6):463–487, 2021. doi:10.1007/s00446-020-00380-5.

7 Keren Censor-Hillel, Orr Fischer, François Le Gall, Dean Leitersdorf, and Rotem Oshman. Quantum distributed algorithms for detection of cliques. In *13th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 215, pages 35:1–35:25, 2022. doi:10.4230/LIPIcs.ITCS.2022.35.

8 Keren Censor-Hillel, Orr Fischer, Ran Gelles, and Pedro Soto. Two for one, one for all: Deterministic ldc-based robust computation in congested clique, 2025. doi:10.48550/arXiv.2508.08740.

9 Keren Censor-Hillel, Orr Fischer, Tzlil Gonen, François Le Gall, Dean Leitersdorf, and Rotem Oshman. Fast distributed algorithms for girth, cycles and small subgraphs. In *34th International Symposium on Distributed Computing (DISC)*, volume 179, pages 33:1–33:17, 2020. doi:10.4230/LIPIcs.DISC.2020.33.

[2] While U_j is a set of wire-values, the Store procedure expects a bitstring, hence we induce some standard order on the set U_j that maps it into a bitstring, and this ordering is known by all nodes.

10 Keren Censor-Hillel, François Le Gall, and Dean Leitersdorf. On distributed listing of cliques. In *Symposium on Principles of Distributed Computing (PODC)*, pages 474–482, 2020. doi:10.1145/3382734.3405742.

11 Keren Censor-Hillel and Einav Huberman. Near-optimal resilient labeling schemes. In *28th International Conference on Principles of Distributed Systems (OPODIS)*, volume 324, pages 35:1–35:22, 2024. doi:10.4230/LIPIcs.OPODIS.2024.35.

12 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 143–152, 2015. doi:10.1145/2767386.2767414.

13 Keren Censor-Hillel and Pedro Soto. Computing in a faulty congested clique. *CoRR*, abs/2505.11430, 2025. doi:10.48550/arXiv.2505.11430.

14 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The complexity of (Δ+1) coloring in congested clique, massively parallel computation, and centralized local computation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 471–480, 2019. doi:10.1145/3293611.3331607.

15 David Cifuentes-Núñez, Pedro Montealegre, and Ivan Rapaport. Recognizing hereditary properties in the presence of byzantine nodes. *CoRR*, abs/2312.07747, 2023. doi:10.48550/arXiv.2312.07747.

16 Sam Coy, Artur Czumaj, Peter Davies, and Gopinath Mishra. Optimal (degree+1)-coloring in congested clique. In *50th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 261, pages 46:1–46:20, 2023. doi:10.4230/LIPIcs.ICALP.2023.46.

17 Artur Czumaj, Peter Davies, and Merav Parter. Simple, deterministic, constant-round coloring in congested clique and MPC. *SIAM J. on Computing*, 50(5):1603–1626, 2021. doi:10.1137/20M1366502.

18 Peter Davies. Optimal message-passing with noisy beeps. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 300–309, 2023. doi:10.1145/3583668.3594594.

19 Danny Dolev, Christoph Lenzen, and Shir Peled. "tri, tri again": Finding triangles and small subgraphs in a distributed setting. In *Distributed Computing*, volume 7611, pages 195–209. Springer, 2012. doi:10.1007/978-3-642-33651-5_14.

20 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 367–376, 2014. doi:10.1145/2611462.2611493.

21 Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 153–162, 2018. doi:10.1145/3210377.3210401.

22 Orr Fischer, Rotem Oshman, and Dana Shamir. Explicit space-time tradeoffs for proof labeling schemes in graphs with small separators. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217, pages 21:1–21:22, 2021. doi:10.4230/LIPIcs.OPODIS.2021.21.

23 Orr Fischer and Merav Parter. Distributed CONGEST algorithms against mobile adversaries. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 262–273, 2023. doi:10.1145/3583668.3594578.

24 Orr Fischer and Merav Parter. All-to-all communication with mobile edge adversary: Almost linearly more faults, for free. *CoRR*, abs/2505.05735, 2025. doi:10.48550/arXiv.2505.05735.

25 Pawel Garncarek, Dariusz R. Kowalski, Shay Kutten, and Miguel A. Mosteiro. Beeping deterministic congest algorithms in graphs. *CoRR*, abs/2502.13424, 2025. doi:10.48550/arXiv.2502.13424.

26 Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 19–28, 2016. doi:10.1145/2933057.2933103.

27 Alex B. Grilo, Ami Paz, and Mor Perry. Distributed non-interactive zero-knowledge proofs. *CoRR*, abs/2502.07594, 2025. doi:10.48550/arXiv.2502.07594.

28 James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–100, 2015. doi:10.1145/2767386.2767434.

29 Taisuke Izumi and François Le Gall. Triangle finding and listing in CONGEST networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 381–389, 2017. doi:10.1145/3087801.3087811.

30 Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2620–2632, 2018. doi:10.1137/1.9781611975031.167.

31 Jonathan Katz and Luca Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 80–86, 2000. doi:10.1145/335305.335315.

32 Janne H. Korhonen. Deterministic MST sparsification in the congested clique. *CoRR*, abs/1605.02022, 2016. doi:10.48550/arXiv.1605.02022.

33 Manish Kumar. Fault-tolerant graph realizations in the congested clique, revisited. In *Distributed Computing and Intelligent Technology*, volume 13776, pages 84–97. Springer, 2023. doi:10.1007/978-3-031-24848-1_6.

34 Manish Kumar, Anisur Rahaman Molla, and Sumathi Sivasubramaniam. Fault-tolerant graph realizations in the congested clique. In *Algorithmics of Wireless Networks*, volume 13707, pages 108–122, 2022. doi:10.1007/978-3-031-22050-0_8.

35 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 42–50, 2013. doi:10.1145/2484239.2501983.

36 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM J. on Computing*, 35(1):120–131, 2005. doi:10.1137/S0097539704441848.

37 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

38 Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

39 Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1096–115, 2020. doi:10.1137/1.9781611975994.67.

40 Krzysztof Nowicki. A deterministic algorithm for the MST problem in constant rounds of congested clique. In *53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1154–1165, 2021. doi:10.1145/3406325.3451136.

41 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the distributed complexity of large-scale graph computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 405–414, 2018. doi:10.1145/3210377.3210409.

42 Merav Parter. (delta+1) coloring in the congested clique model. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 107, pages 160:1–160:14, 2018. doi:10.4230/LIPIcs.ICALP.2018.160.

43 Merav Parter and Hsin-Hao Su. Randomized (Delta+1)-coloring in O(log* Delta) congested clique rounds. In *32nd International Symposium on Distributed Computing (DISC)*, volume 121, pages 39:1–39:18, 2018. doi:10.4230/LIPIcs.DISC.2018.39.

44 David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.

45 Daniel A. Spielman. Highly fault-tolerant parallel computation. In *Proceedings of 37th Conference on Foundations of Computer Science (FOCS)*, pages 154–163, 1996. doi:10.1109/SFCS.1996.548474.

46 Sergey Yekhanin. Locally decodable codes. *Foundations and Trends® in Theoretical Computer Science*, 6(3):139–255, 2012. doi:10.1561/0400000030.

A Chernoff Bounds

▶ **Theorem 15** (Chernoff inequality for independent Bernoulli variables). *Let $X_1, \ldots, X_n$ be mutually independent 0–1 random variables with $\Pr(X_i = 1) = p_i$. Let $X = \sum_{i=1}^{n} X_i$ and set $\mu = E[X]$. The following holds,*

1. *for any $\delta > 0$, $\Pr(X \geq (1+\delta)\mu) \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$*
2. *for $0 < \delta \leq 1$, $\Pr(X \geq (1+\delta)\mu) \leq e^{-\mu\delta^2/3}$*
3. *for $R \geq 6\mu$, $\Pr(X \geq R) \leq 2^{-R}$*

For proof, see Theorem 4.4 in [38].

B Missing Proofs from Section 3

Proof of Lemma 12. Set $r = |G|$. let $g_1, \ldots, g_r$ be the gates in G sorted in a decreasing order of their fan. For any $i \in [r]$, set $d_i = \mathsf{fan}(g_i)$, then,

$$d_r \leq d_{r-1} \leq \ldots \leq d_1 \leq \Delta.$$

First, we analyze the case where $|\mathcal{A}|/2 \leq L \leq |\mathcal{A}|$. We notice that the load of each vertex v is trivially bounded by $\lambda(v) \leq \sum_{i=1}^{r} d_i$ (which is obtained if all gates are allocated to v). On the other hand, by assumption that $|\mathcal{A}|/2 \leq L$ it follows that

$$\lambda(v) \leq \sum_{i=1}^{r} d_i = P \leq 2PL/|\mathcal{A}|,$$

and the claim for the case of $|\mathcal{A}|/2 \leq L \leq |\mathcal{A}|$ follows.

We assume in the remainder of the proof that $L < |\mathcal{A}|/2$. First, we make the very simple observation that at all times during the procedure, it holds that

$$\sum_{v \in V} \lambda(v) \leq \sum_{i=1}^{r} d_i \cdot L = PL. \tag{1}$$

We split the analysis into two phases of the greedy allocation procedure: we say that the procedure is in its first phase while $d_i \geq 2PL/|\mathcal{A}|$ and we say that it is in its second phase while $d_i < 2PL/|\mathcal{A}|$. In each phase, we show that following a gate being allocated to batch of L vertices, all vertices have load at most $\max(4PL/|\mathcal{A}|, \Delta)$.

While $d_i \geq 2PL/|\mathcal{A}|$, we must have at least $|\mathcal{A}|/2$ vertices with no load. Assume otherwise, then we notice that since the gates $g_1, \ldots, g_r$ are sorted in descending order according to $d_1, \ldots, d_r$, then each vertex with an allocated gate has load at least $d_i \geq 2PL/|\mathcal{A}|$. Summing the loads of all vertices, we conclude that the total load is at least $\frac{2PL}{|\mathcal{A}|}(\frac{|\mathcal{A}|}{2} + 1) > PL$, contradicting Equation (1). Hence, after any allocation where $d_i \geq 2PL/|\mathcal{A}|$, we only pick vertices with $\lambda(v) = 0$, and the load of any such vertex v is therefore at most Δ.

Next, we consider the phase where $d_i \leq 2PL/|\mathcal{A}|$. By an averaging argument on Equation (1), at any point of time of the procedure we have at least $|\mathcal{A}|/2$ vertices with load at most $2PL/|\mathcal{A}|$. Since $L < |\mathcal{A}|/2$, the new load of a vertex v that is assigned a new gate is at most

$$\lambda(v) \leq d_i + \frac{2PL}{|\mathcal{A}|} \leq \frac{4PL}{|\mathcal{A}|},$$

and the claim follows. ◀