מכון ויצמן למדע
WEIZMANN INSTITUTE OF SCIENCE

Thesis for the degree
Doctor of Philosophy

עבודת גמר (תזה) לתואר
דוקטור לפילוסופיה

Submitted to the Scientific Council of the
Weizmann Institute of Science
Rehovot, Israel

מוגשת למועצה המדעית של
מכון ויצמן למדע
רחובות, ישראל

By
Yael Hitron

מאת
יעל חיטרון

אלגוריתמים מבוזרים עם תקשורת רועשת
Distributed computation with noisy communication

Advisor:
Professor Merav Parter

מנחה:
פרופסור מרב פרטר

November 2023

כסלו תשפ"ד

Acknowledgments

I am very happy to begin the Acknowledgments section by thanking my advisor, Merav Parter. Merav, your support, our open communication, and the belief you have in me mean a great deal to me. Even during the busiest days, your door was always open to help out with some technical difficulties, discuss science, or any other topic that was on my mind. Your consistent optimism and love for graphs and research problems has been truly inspiring. I am very fortunate to have had the opportunity to work and learn from you.

I would also like to thank my PhD committee, Guy Rothblum and Uri Feige, for their time, wisdom, and good advice. I especially thank Uri Fiege who was also my MSc advisor for his encouragement, and for introducing me to the world of research. Special thanks to Itai Benjamini for his mentorship and for recommending Merav as my advisor.

During my PhD, I had the opportunity to work with wonderful collaborators and co-authors: Gur Perry, Cameron Musco, Nancy Lynch, and Eylon Yogev. It was a pleasure working with you. I would like to thank the theory group and its affiliates: Uriel Feige, Robert Krauthgamer, Amir Abboud, Moni Naor, Irit Dinur, Oded Goldreich, Guy Rothblum, Zvika Brakerski, Shahar Dobzinski, and David Peleg. Thank you for creating an inspiring and nurturing research environment.

I truly enjoyed sharing this journey with our research group over the years: Noa Shahar, Gur Perry, Uri Ben-Levy, Asaf Petruschka, Orr Fischer, and Shimon Kogan. Thank you for all the discussions, brainstorming, and friendship. My time at the Weizmann Institute was extremely pleasant also due to the many good friends and office mates I met. The list is long, and I am bound to forget someone. Still, I will try to thank the following (possibly partial) people personally: Dan Mikulincer, Gil Gofer, Chen Amiraz, Avi Cohen, Havana Rika, Yotam Dikstein, Gal Yona, Ori Katz, Gilad Yehudai, Odelia Melamed, Hila Dahari, Shiri Ron, Yuval Belfer, Heli Ben Hemu, Shay Sapir, Ariel Shaulker, Lior Yariv, Elad Tzalic, and Guy Kornowsky.

Lastly, this section cannot be complete without thanking my family, who provide the support behind everything I do. To my parents Orna and Rony, and to my husband Nitay, who have stood by me throughout this chaotic period of deadlines, distant conferences, and late-night working sessions. And to my daughters Noga and Yarden, who bring me so much joy and inspiration.

Declaration

I hereby declare that this thesis summarizes my original research, performed under the guidance of my advisor Prof. Merav Parter. Other than that, the work presented in Chapter 3 was done in collaboration with Prof. Cameron Musco.

Yael Hitron

Contents

Abstract

A distributed network is composed of a collection of interconnected entities that collaborate and communicate with each other to achieve a shared goal. The communication over distributed networks is often subject to either random or adversarial noise. In this thesis, we study distributed networks with noisy communication from two perspectives. In the first part of the thesis, we focus on stochastic noise in biologically inspired neural networks. We consider the model of spiking neural networks (SNN) [129, 128], a simple yet biologically plausible model that captures the spiking behavior observed in real neural networks. In this setting, we focus on two results. First, we study algorithmic aspects of time measurement and counting. Second, we study biological neural networks from the perspective of streaming algorithms, establishing novel connections between the two models.

The second part of the thesis considers adversarial noise, where we study distributed algorithms that are resilient against adversarial attacks. We focus on the classical CONGEST model of distributed computing [164], in which the network is represented as an undirected graph, and the vertices communicate in synchronous rounds with messages of limited size. We consider a computational-unbounded adversary that corrupts the computation by sending malicious messages over (apriori unknown) controlled edges. In this setting, we start with studying the broadcast problem, where a designated source vertex wishes to send a message to all other vertices in the graph. We then turn to a more general framework and provide a compiler that simulates any CONGEST algorithm in the adversarial setting.

Introduction

The main focus of this thesis is the study of distributed computation with noisy communication. Essentially, a distributed network is a collection of interconnected entities (computers, agents, etc.) that communicate and collaborate to perform a computation for the entire network. Distributed networks exhibit pervasive characteristics and capture a diverse range of computational settings. Among the many forms of distributed networks are computer networks, servers in data centers, ant colonies, and neural networks within the brain. In real-world scenarios, distributed networks are often subject to disruptions and noise. There are two common sources of noise that can significantly impact the networks computation: stochastic and adversarial. Stochastic noise refers to random fluctuations that arise naturally in the network due to various factors such as signal interference, environmental conditions, or inherent variability in the system components. Stochastic noise appears naturally in biological systems such as the human brain. Adversarial noise, on the other hand, consists of malicious entities intentionally interfering with the normal flow of information within the network. Adversarial noise can be strategically designed to exploit vulnerabilities and compromise the computation's correctness.

The primary goal of this work is to develop distributed networks and algorithms in noisy distributed models. Taking into account the natural characteristics of each noise type, we consider two distinct distributed settings. The first part of the thesis focuses on stochastic noise in biological neural networks. We study the biologically inspired model of spiking neural networks from the point of view of distributed algorithms. Since neuronal spikes are subject to probabilistic disturbances, stochastic noise plays an important role in this model. The second part of the thesis focus on adversarial noise. Specifically, we study distributed algorithms that are resilient against adversarial edge corruption in the classical CONGEST model. We note that while the first part of this thesis leverages from stochastic noise, the second part counteracts the adversarial noise which threatens the correctness of the computation. The rest of this chapter elaborates on each part separately, and briefly summarizes our main results.

Spiking Neural Networks From an Algorithmic Perspective

We consider resilient distributed computation in one of the most fascinating biological systems – networks of spiking neurons like those found in the human brain. The human brain is a complex network performing a variety of computational tasks. Understanding how the brain works as a computational device is a central challenge of modern neuroscience and artificial intelligence. Different research communities approach this task in different ways, including functional imaging that studies neural activation patterns, examining neural network structures as a clue to computational function, and engineering neural-inspired machine learning architectures. We study neural networks from the perspective of distributed networks and the theory of computation in general.

We consider a simple yet biologically plausible model of *spiking neural networks* (SNN) [128, 129, 96]. In this model, neurons fire in discrete pulses, in response to a sufficiently high membrane potential. This potential is induced by spikes from neighboring neurons, which can have an excitatory or inhibitory effect, either increasing or decreasing the potential. The neurons then spike with a sigmoidal probability that depends on their membrane potential. In this part of the thesis, we highlight two results. The first work studies time measurement and counting tasks in spiking neural networks. In the second work we study the connection between spiking neural networks and streaming algorithms.

Related Work. Neural networks have been studied in several academic communities, from varying perspectives. Significant work in computational neuroscience focuses on developing somewhat realistic mathematical models for neural networks and studying their capacity to process information [96, 185, 74, 175]. On the more theoretical side, a variety of artificial network models such as perceptrons, sigmoidal networks, Hopfield networks and Boltzmann machines have been developed [54]. These models are tractable for theoretical analysis, and have been studied in the context of their computational power, and their ability to solve problems of function approximation, classification, and memory storage [90, 133, 178, 129]. In practical machine learning, biological fidelity and often theoretical tractability are set aside, and researchers study how neural-like networks can be used to efficiently represent and learn complex concepts [83, 110]. Mass at el. considered the computational power of spiking neural networks [129, 130, 131], whereas Valiant [189, 190, 191] defined models of computation and investigated implementations of basic learning tasks within these models. The algorithmic aspects of spiking neural networks have recently received quite a lot of attention in the algorithmic community [119, 124, 123, 112, 148, 44, 132, 120, 6, 151, 45, 7, 177, 147, 123]. The goal of this thesis is to further expand this line of research by exploring the algorithmic aspects of a number of fundamental problems in SNN, as well as generalizing their connections to different models in theoretical computer science.

Main Results

We next give a brief overview of each chapter in this section.

Time measurement and compressed counting. In Chapter 2 (based on [86]), we consider the algorithmic aspects of measuring time in the model of spiking neural networks. Discovering the underlying mechanisms by which the brain perceives the duration of time is one of the largest open enigmas in computational neuroscience. Humans measure time using a global clock based on standardized units of minutes, days, and years. In contrast, the brain perceives time using specialized neural clocks that define their own time units.

In this work, first we consider a deterministic setting where a neuron fires in a given round if the membrane potential exceeds a fixed threshold. In this setting, we show a network construction that can measure t time units with an optimal number of $O(\log t)$ neurons. Another goal of this work is to understand the power and limitations of stochastic noise in neural networks. Neural computation in general, and neural spike responses in particular, are inherently stochastic. We therefore consider a randomized variant of the problem in a stochastic setting where each neuron spikes randomly, with probability determined by its membrane potential, and the goal is to count $O(t)$ time units. This randomized variant led to an improved solution using $O(\log\log t)$ neurons, providing the first theoretical proof that randomization (i.e., noise) can in fact facilitate neural computation[1].

Additionally, we consider a counting variant of the problem where we wish to count efficiently the number of times a neuron spikes during a given period of time in a compressed manner. Spiking neurons are believed to encode information via their firing rates. This underlies the rate coding scheme in which the spike count of the neuron in a given period is interpreted as a letter in a larger alphabet. In a network of memory-less spiking neurons, it is not so clear how to implement this rate-dependent behavior. We show how the deterministic timer can be modified to count the number of times a neuron fired in a time span of t rounds using $O(\log t)$ many additional neurons.

As for neural counting in the randomized setting, the problem of maintaining a counter using a small amount of space has received a lot of attention in the dynamic streaming community. The well-known Morris algorithm [70, 140] maintains an approximate counter for t counts using only an order of $\log\log t$ bits. By following ideas from [70], carefully adapted to the neural setting, we show a neural construction for approximate counting in a compressed manner using an order of $\log\log t$ additional neurons. The approximate counting problem provides just one indication of the relation between succinct neural networks and dynamic streaming algorithms, as shown in Chapter 3.

Finally, we demonstrate the usefulness of compressed counting and timer networks for synchronizing neural networks. In the spirit of distributed synchronizers [15], we provide a general transformation that can take any synchronized network solution and simulate it in an asyn-

[1]We note that in the deterministic setting solving this variate also requires at least $\Omega(t)$ neurons.

chronous setting (where edges have arbitrary response latencies) while incurring a small overhead w.r.t the number of neurons and computation time.

Connection to streaming algorithms. In Chapter 3 (based on [85]), we study biological neural networks from the perspective of streaming algorithms. Like computers, human brains suffer from memory limitations which pose a significant obstacle when processing large-scale and dynamically changing data. In computer science, these challenges are captured by the well-known streaming model and have had a significant impact in theory and beyond. In the classical streaming model [142], one must compute some function f of a stream of updates, given restricted single-pass access to the stream. The primary complexity measure is the space used by the algorithm. In contrast to the large body of work on streaming algorithms, relatively little is known about the computational aspects of data processing in biological neural networks. In this work, we aim at connecting these two models, leveraging techniques developed for streaming algorithms to better understand neural computation.

Previous algorithmic work in spiking neural networks has many similarities with streaming algorithms. In the study of spiking neural networks, space-efficient SNNs have been devised for the winner-takes-all problem [118, 182], similarity testing and compression [124, 148], clustering [84, 112], approximate counting, and time estimation [120, 86]. Interestingly, many of these works borrow ideas from related streaming algorithms. However, despite the flow of ideas from streaming to neural algorithms, the connection between these models has not been studied formally.

In this work, we take the first steps toward understanding this connection. On the upper bound side, we design neural algorithms based on known streaming algorithms for fundamental tasks, including distinct elements, approximate median, and heavy hitters. The number of neurons in our neural solutions almost matches the space bounds of the corresponding streaming algorithms. As a general algorithmic primitive, we show how to implement the important streaming technique of linear sketching efficiently in spiking neural networks. On the lower bound side, we give a generic reduction, showing that any space-efficient spiking neural network can be simulated by a space-efficiently streaming algorithm. This reduction lets us translate streaming-space lower bounds into nearly matching neural-space lower bounds, establishing a close connection between these two models.

Distributed Computing Against Adversarial Edges

Guaranteeing the uninterrupted operation of communication networks is a significant objective in network algorithms. The area of resilient distributed computation has been receiving a growing attention over the last years as computer networks grow in size and become more vulnerable to adversarial failures. In this part of the thesis, we focus on resilience against adversarial edge corruptions in the classical CONGEST model of distributed computing [164]. In this model, distributed systems are modeled as graphs with n vertices connected by communication links. The computation proceeds in synchronous rounds, in which every vertex can send a message of size $O(\log n)$ bits to each of its neighbors. The main complexity measure considered in this model is the round complexity.

In the *adversarial* CONGEST model, an adversary with unbounded computational power controls a fixed set of edges in the graph (unknown to the vertices). It is allowed to see the entire graph, the messages sent throughout the algorithm, and the internal randomness of the vertices. within this setting, our goal is to provide efficient distributed algorithms for fundamental computational tasks. These algorithms are required to preserve the correctness of the computation (despite the adversarial attack) while being also efficient in terms of the number of rounds.

A key limitation of many of the previous resilient algorithms is that they assume that the communication graph is the complete graph. In this research direction, we concentrate on communication graphs of arbitrary topologies. We began with studying the corner-stone broadcast problem in which a designated source vertex wishes to send a message to all the other vertices in the graph. We then turn to a more general framework, and provided a general simulation methodology that compiles any (reliable) CONGEST algorithm $\mathcal{A}$ into an equivalent algorithm $\mathcal{A}'$ in the presence of adversarial edges.

Related Work. Since the introduction of the adversarial setting by Pease et al. [159] and Lamport et al. [109, 159] distributed algorithms against various adversarial models have been studied in theory and practice. In most resilient distributed algorithms, the communication graph is assumed to be a complete graph e.g., [24, 47, 57, 58, 63, 103, 137, 162, 75, 159], and little is known about the complexity of resilient computations in general graph topologies. See [167] for an overview. In their seminal works, Dolev and Pelc [57, 160] showed that any given graph can tolerate up to f adversarial vertices or edges iff the graph is $(2f+1)$ vertex or edge connected. Unfortunately, the existing distributed algorithms for general $(2f+1)$ connected graphs usually require a polynomial number of rounds in the CONGEST model. Other than that, resilient algorithms for general graph topologies have been addressed mostly under simplified settings [162], e.g., probabilistic faulty models [163, 160], cryptographic assumptions [73, 2, 1, 145, 32, 1, 19, 18], or under bandwidth-*free* settings (e.g., allowing neighbors to exchange exponentially large messages) [57, 137, 105, 107, 59, 105, 43].

One exception is the work of Parter and Yogev [158, 156, 155]. Motivated by various applications for resilient distributed computing, Parter and Yogev introduced the notion of *low-congestion cycle covers* as a basic communication backbone for reliable communication [155]. At the high level, a low-congestion cycle cover is a collection of cycles covering all the edges in the graph, where all cycles are both short and nearly edge-disjoint. [155] proved the existence of cycle covers in bridgeless graphs, and demonstrate their usefulness in resilient computation. Specifically, they show that by computing these cycle covers in a pre-processing step, one can compile any algorithm in the (fault-free) CONGEST model into an equivalent algorithm that is resilient against a single adversarial edge. In chapter 6 we extend their simulation methodology to handle multiple edge faults and eliminate the need for fault-free preprocessing.

Main Results

We next give an overview of the main results presented in each chapter.

The broadcast problem. In Chapter 5 (based on [87]), we consider the corner-stone broadcast problem in the adversarial CONGEST model, where an adaptive adversary controls a fixed number of f edges in the input communication graph. In this work, we provide the first round-efficient broadcast algorithms against edge adversaries. Our approach is based on combining the perspectives of fault tolerant (FT) network design, and distributed graph algorithms. The combined power of these points of view allows us to characterize the round complexity of resilient broadcast algorithms as a function of the graph diameter D, and the number of adversarial edges f. This is in contrast to prior algorithms that obtain a polynomial round complexity (in the number of vertices).

We take a gradual approach and start by studying broadcast algorithms against a single adversarial edge. In this setting, we present a deterministic algorithm that solves the problem within $\widetilde{O}(D^2)$ rounds[1], provided that the graph is 3 edge-connected. This improves considerably upon the (implicit) state-of-the-art $n^{O(D)}$ bound obtained by previous algorithms (e.g., by [137, 43]). In addition, in contrast to many previous works (including [137, 43]), our algorithm does not assume global knowledge of the graph or any estimate on the graph's diameter. In fact, at the end of the broadcast algorithm, the vertices also obtain a linear estimate of the graph diameter. We then extend this algorithm to a $\widetilde{O}(D^{O(f)})$-round algorithm against f adversarial edges in $(2f+1)$ edge-connected graphs.

Next, we turn to consider the family of expander graphs, which has been shown to have various applications in the context of resilient distributed computation [61, 187, 106, 12]. For expander graphs with conductance ϕ and minimum degree of $\Theta(f^2 \log n/\phi)$, we give a considerably improved broadcast algorithm with $O(f \log^2 n/\phi)$ rounds against $f = O(\frac{n \cdot \phi}{\log n})$ adversarial edges. This algorithm exploits the connectivity and conductance properties of G-subgraphs obtained by employing Karger's edge sampling technique [100].

General Compiler. In Chapter 6 (based on [88]), we present a compilation methodology that converts any given distributed algorithm $\mathcal{A}$ in the standard CONGEST model into an equivalent algorithm $\mathcal{A}'$ that can perform the same computation in the presence of adversarial edges. The compilation process involves translating each round of the fault-free algorithm into a phase of multiple rounds in the resilient algorithm. The goal is to ensure that by the end of each phase, all vertices hold the correct messages sent by their neighbors in that round, while disregarding

[1]The notation $\widetilde{O}(\cdot)$ hides poly-logarithmic terms in the number of vertices n.

any remaining corrupted messages. The main complexity measure of the compilation process is the round complexity of compiling a single-round, which represents the *compilation overhead* of the compiler.

In order to handle multiple adversarial edges, we introduce the concept of a fault-tolerant (FT) cycle cover, which extends the low-congestion cycle cover of [155]. An f-FT cycle cover is a collection of cycles that covers all the edges of the graph, such that for any set of at most f edges F and every edge e, there exists a cycle that covers e while avoiding the edges in F. This guarantees that every edge is covered by a reliable cycle, avoiding all f adversarial edges. We start with exploring the combinatorial properties of FT cycle covers, and then focus on constructing FT cycle covers in the presence of adversarial edges. The main challenge in constructing FT cycle covers in the adversarial setting is covering the unknown adversarial edges.

Given a FT cycle cover construction, we show general compilation algorithms in the adversarial CONGEST model. Initially, we consider graphs with a single adversarial edge. For every 3 edge-connected graph with a diameter of D, we demonstrate the existence of a general compiler against a single adversarial edge, achieving a compilation overhead of $\widehat{O}(D^3)$ rounds[1]. This improvement surpasses the $\widehat{O}(D^5)$ round overhead obtained by Parter and Yogev [154], while eliminating the need for a fault-free preprocessing phase. Subsequently, we extend our algorithms to graphs with multiple adversarial edges. For any $(2f+1)$ edge-connected graph G, we present a general compiler against f adversarial edges, achieving a compilation overhead of $\widehat{O}(D^{O(f)})$ rounds.

[1]The notation $\widehat{O}(.)$ hides factors of $2^{O(\sqrt{\log n})}$, which arise from the distributed algorithms of [156, 157].

PART I

ALGORITHMIC NEURAL NETWORKS

1

Background and Preliminaries

The study of neural networks is conducted in a variety of academic communities from a variety of perspectives. Several studies in computational neuroscience focuses on understanding how the brain encodes and decodes information, delving into the mechanisms through which sensory neurons translate stimulus information into action potentials, or spikes. This involves analyzing the collective activity patterns of neural populations using statistical tools, information theory, and new mathematical models applied to real experimental data [175, 74, 76, 166, 144, 176]. In this line of work, it has been highlighted that the combined activity of a neuronal cluster offers more insights than merely considering individual neuronal activities, emphasizing the pivotal role of neural correlations in processing information.

This research exemplifies the symbiotic relationship between neuroscience and computational methods. While neuroscience leans on computational techniques to analyze and integrate large data sets, it conversely provides valuable inspiration for computational algorithms. One example is the work of Afek et al. [5, 4], who drew connections between a task performed in the fruit fly brain and the maximal independent set (MIS) problem. Leveraging from biological findings on processes in developing flies, they designed a computational algorithm for MIS selection that improves the efficiency and robustness of existing methods. Similarly, Dasgupta et al. extrapolated principles from the fly's olfactory system to devise novel solutions to computational problems such as the nearest-neighbor search, and tasks like online image searching [52]. In a followup work [51], they show that the fly's olfactory system also utilizes a variant of a Bloom filter, leading to improved Bloom filters, better suited for various computational and biological datasets. These instances demonstrate how studying biological neural networks can significantly advance computational methodologies.

Another research area that utilizes biological insights to improve computational abilities is Neuromorphic computing. Inspired by the structure and function of the human brain, Neuromorphic computing aims to develop more efficient and powerful computing systems by emulating how neurons and synapses operate. These systems are designed to mimic the behavior of individual neurons, which communicate through discrete electrical pulses or spikes. By using networks of neurons, neuromorphic systems can achieve a higher level of energy efficiency compared to traditional computing architectures. This is because spikes occur only when there is relevant information to be processed, reducing unnecessary power consumption. In practice, both academic and industrial initiatives are working to scale up neuromorphic systems to include large numbers of neurons [139, 53, 104, 22, 172]. See [136, 72] for surveys on the topic. From a theoretical point of view, a recent work focused on the design and analysis of neuromorphic algorithms based on recurrent neural networks, focusing on shortest path problems [6]. In their work, they developed comparison tools to compare neuromorphic algorithms with traditional computing, and demonstrated the resource advantages of the suggested neuromorphic solutions.

In recent years, computational methods have been used to explore the brain's formation of structures and concepts found in biological experiments. The focus of these studies is on neural networks with plasticity rules, in which synaptic strength, or edge weights, adjust over time due to neural activity. As an example, one line of research examined the formation of assemblies of neurons, which play an important role in memory traces for spatial information and real-world items. This research studies the emergence and modification of neuronal assemblies, as well as operations on assemblies, including projections, associations, and merges [50, 149, 141, 148, 147, 165, 112]. In another study, hierarchical structures have been studied, along with how brain-like neural networks might be used to represent them, how they might be used to recognize concepts, and how they might be learned[122, 121].

In this thesis, we consider a computational model for neural computation that balances biological plausibility with theoretical tractability. Specifically, we focus on *spiking neural networks* (SNNs) [128, 129, 96], in which a neuron fires in discrete pulses, in response to a sufficiently high membrane potential. This potential is induced by spikes from neighboring neurons, which can have an excitatory or inhibitory effect, either increasing or decreasing the potential. The study of spiking neural networks was initiated by Maass, focusing on their computational capabilities. In his research, he showed general constructions in the SNN model that simulate a variety of computational paradigms, such as boolean circuits, finite automata, Turing machines, and random access machines [126, 128]. More generaly, Maass also drew comparisons between SNNs and other computational models, including sigmoidal networks and boolean threshold gates, emphasizing the computational advantages of SNNs [133, 127, 129].

Diverging from conventional approaches in computational neuroscience that emphasize general computation ability or broad learning tasks, our research builds upon the groundwork laid by Parter, Lynch, and Musco, who focused on the algorithmic aspects of spiking neural networks [123, 118, 124, 119]. This research direction is aimed at studying a wide range of fundamental computational problems solved by actual brains. This includes problems such as winner takes all, similarity testing, and compression. For these problems, they construct efficient biologically–plausible algorithms, consider lower bounds on the size of the network and computational time, and study the tradeoffs between the resources of the network.

In this thesis we cover work originally published in [86, 85]. In Chapter 2 [86], we consider the implementation of core algorithmic primitives within neural computation, such as time perception, counting, and synchronization, with a comprehensive analysis of time and space complexities. We also illustrate how the neuronal stochastic behavior can be exploited to optimize space complexity. In Chapter 3 [85], we study the relationship between SNNs and streaming algorithms which process large data streams in a single pass with limited memory. As a result of these connections, SNN upper and lower bounds can be obtained using results obtained in the streaming model.

In a related study [84], we explored the execution of tasks such as clustering, novelty detection, and sketching, which are evident in natural systems like the olfactory process in fruit flies [35, 116, 52]. In a separate research [89], we delved deeper into the synchronization dynamics in SNNs, building on the foundation laid in [86]. In this work, we expanded the SNN model to asynchronous settings by incorporating edge and vertex delays. We also study general synchronization schemes and their cost in terms of the overhead in the number of neurons and computation time.

The Computational Model

We consider the model of *spiking neural networks* (SNNs) [128, 129, 96] defined as follows.
Neurons. A deterministic neuron u is modeled by a *deterministic* threshold gate. Letting $b(u)$ be the threshold value of u, then u outputs 1 if the weighted sum of its incoming neighbors exceeds the threshold $b(u)$. A *spiking neuron* is modeled by a probabilistic threshold gate which fires with a sigmoidal probability that depends on the difference between its weighted incoming sum and $b(u)$.

Spiking neural networks. A Spiking neural network (SNN) $\mathcal{N} = \langle X, Z, Y, w, b\rangle$ consists of n input neurons $X = \{x_1, \ldots, x_n\}$, m output neurons $Y = \{y_1, \ldots, y_m\}$, and k auxiliary neurons $Z = \{z_1, ..., z_k\}$. In a *deterministic* neural network all neurons are deterministic threshold gates. In *randomized* neural network, the neurons can be either deterministic threshold gates or probabilistic threshold gates. The directed weighted synaptic connections between the neurons $V = X \cup Z \cup Y$, are described by the weight function $w : V \times V \to \mathbb{R}$. A weight $w(u, v) = 0$ indicates that a connection is not present between neurons u and v. Finally, for any neuron v, the value $\mathrm{b}(v) \in \mathbb{R}$ is the threshold value (activation bias). The in-degree of every input neuron x_i

is zero, i.e., $w(u,x)=0$ for all $u \in V$ and $x \in X$. Additionally, each neuron is either inhibitory or excitatory: if v is inhibitory, then $w(v,u) \leq 0$ and if v is excitatory, then $w(v,u) \geq 0$ for every u.

Network dynamics. The network evolves in discrete, synchronous rounds as a Markov chain. The firing probability of every neuron in round τ depends on the firing status of its neighbors in round $\tau-1$, via a standard sigmoid function, with details given below. For each neuron u, and each round $\tau \geq 0$, let $\sigma_\tau(u)=1$ if u fires (i.e., generates a spike) in round τ. Let $\sigma_0(u)$ denote the initial firing state of the neuron. The firing state of each input neuron x_j in each round is the input to the network. For each non-input neuron u and every round $\tau \geq 1$, let $\mathrm{pot}(u,\tau)$ denote the membrane potential at round τ and $p(u,\tau)$ denote the firing probability ($\Pr[\sigma_\tau(u)=1]$), calculated as $\mathrm{pot}(u,\tau)=\sum_{v \in V} w(v,u) \cdot \sigma_{\tau-1}(v) - b(u)$ and $p(u,\tau)=\frac{1}{1+e^{-\mathrm{pot}(u,\tau)/\lambda}}$ where $\lambda > 0$ is a *temperature parameter* which determines the steepness of the sigmoid. λ does not affect the computational power of the network, thus we set $\lambda = 1$.

2

Counting to Ten with Two Fingers: Compressed Counting with Spiking Neurons

2.1 Introduction

Understanding the mechanisms by which the brain experiences time is one of the major research objectives in neuroscience [138, 8, 64]. Humans measure time using a global clock based on standardized units of minutes, days, and years. In contrast, the brain perceives time using specialized neural clocks that define their own time units. Living organisms have various other implementations of biological clocks, a notable example is the circadian clock that gets synchronized with the rhythms of a day.

In this work, we consider the algorithmic aspects of measuring *time* in a simple yet biologically plausible model of *stochastic spiking neural networks* (SNN) [128, 129], in which neurons fire in discrete pulses, in response to sufficiently high membrane potential. This model is believed to capture the spiking behavior observed in real neural networks and has recently received quite a lot of attention in the algorithmic community [118, 119, 124, 123, 112, 148, 44]. In contrast to the common approach in computational neuroscience and machine learning, the focus here is not on general computation ability or broad learning tasks, but rather on specific algorithmic implementation and analysis.

2.1.1 Measuring Time with Spiking Neural Networks

We consider the algorithmic challenges of measuring time using networks of threshold gates and probabilistic threshold gates. We introduce the *neural timer* problem defined as follows:

> Given an input neuron x, an output neuron y, and a time parameter t, it is required to design a small neural network such that any firing of x in a given nd invokes the firing of y for exactly the next t rounds.

In other words, it is required to design a succinct timer, activated by the firing of its input neuron, that alerts when exactly t rounds have passed.

A trivial solution with t auxiliary neurons can be obtained by taking a directed chain of length t (Fig. 2.1): the head of the chain has an incoming edge from the input x, the output y has incoming edges from the input x, and all the other t neurons on the chain. All these neurons are simple OR-gates, they fire in round τ if at least one of their incoming neighbors fired in round $\tau - 1$. Starting with the firing of x in round 0, in each round i, exactly one neuron, namely the i-th neuron on the chain fires, which makes y keep on firing for exactly t rounds until the chain fades out. In this basic solution, the network spends one neuron that counts $+1$ and dies. It is noteworthy that the neurons in our model are very simple, they do not have any memory, and thus cannot keep track of the firing history. They can only base their firing decisions on the firing of their neighbors in the *previous* round.

With such a minimal model of computation, it is therefore intriguing to ask how to beat this linear dependency (of network size) in the time parameter t. Can we count to ten using only two (memory-less) neurons? We answer this question in the affirmative and show that even with just simple deterministic threshold gates, we can measure time up to t rounds using only $O(\log t)$ neurons. It is easy to see that this bound is tight when using deterministic neurons (even when allowing some approximation). The reason is that $o(\log t)$ neurons encode strictly less than t distinct configurations, thus in a sequence of t rounds, there must be a configuration that re-occurs, hence locking the system into a state in which y fires forever.

Theorem 1 (Deterministic Timers). *For every input time parameter $t \in \mathbb{N}_{>0}$, (1) there exists a deterministic neural timer network $\mathcal{N}$ with $O(\log t)$ deterministic threshold gates, (2) any deterministic neural timer requires $\Omega(\log t)$ neurons.*

This timer can be easily adapted to the related problem of *counting*, where the network should output the number of spikes (by the input x) within a time window of t rounds.

Does randomness help in time estimation? Neural computation in general, and neural spike responses in particular, are inherently stochastic [117]. One of our broader scope agendas is to understand the power and limitations of randomness in neural networks. Does neural computation become *easier* or *harder* due to the stochastic behavior of the neurons?

We define a randomized version of the neural timer problem that allows some slackness both in the approximation of the time, as well as allowing a small error probability. For a given error probability $\delta \in (0,1)$, the output y should fire for at least t rounds and must stop firing after at most $2f$ rounds[1] with probability at least $1-\delta$. It turns out that this randomized variant leads to a considerably improved solution for $\delta = 2^{-O(t)}$:

Theorem 2 (Upper Bound for Randomized Timers)**.** *For every time parameter $t \in \mathbb{N}_{>0}$, and error probability $\delta \in (0,1)$, there exists a probabilistic neural timer network $\mathcal{N}$ with $O(\min\{\log\log 1/\delta, \log t\})$ deterministic threshold gates and a* single *random spiking neuron.*

Our starting point is a simple network with $O(\log 1/\delta)$ neurons, each firing independently with probability $1-1/t$. The key observation for improving the size bound into $O(\log\log 1/\delta)$ is to use the *time axis*: we will use a *single* neuron to generate random samples over time, rather than having *many* random neurons generating these samples in a *single* round. The deterministic neural counter network with a time parameter of $O(\log 1/\delta)$ is used as a building block to gather the firing statistics of a single spiking neuron. In light of the $\Omega(\log t)$ lower bound for deterministic networks, we get the first separation between deterministic and randomized solutions for error probability $\delta = \omega(1/2^t)$. This shows that randomness can help, but up to a limit: Once the allowed error probability is exponentially small in t, the deterministic solution is the best possible. Perhaps surprisingly, we show that this behavior is tight:

Theorem 3 (Lower Bound for Randomized Timers)**.** *Any SNN for the (randomized) neural timer problem with time parameter t, and error $\delta \in (0,1)$ must use $\Omega(\min\{\log\log 1/\delta, \log t\})$ neurons.*

Neural counters. Spiking neurons are believed to encode information via their firing rates. This underlies the *rate coding* scheme [3, 186, 77] in which the spike-count of the neuron in a given period is interpreted as a *letter* in a larger alphabet. In a network of memory-less spiking neurons, it is not so clear how to implement this rate-dependent behavior. How can a neuron convey a complicated message over time if its neighboring neurons remember only its recent spike? This challenge is formalized by the following neural counter problem: Given an input neuron x, a time parameter t, and $\Theta(\log t)$ output neurons represented by a vector $\bar{y}$, it is required to design a neural network such that the output vector $\bar{y}$ holds the binary representation of the number of times that x fired in a sequence of t rounds. This problem is very much related to the neural timer problem and can be solved deterministically using $O(\log t)$ neurons. Can we do better?

The problem of maintaining a *counter* using a small amount of space has received a lot of attention in the *dynamic streaming* community. The well-known Morris algorithm [140, 70] maintains an approximate counter for t counts using only $O(\log\log t)$ bits. The high-level idea

[1]Taking $2f$ is arbitrary here, and any other constant greater than one would work as well.

of this algorithm is to increase the counter with a probability of $1/2^C$ where C is the current read of the counter. The counter then holds the exponent of the number of counts. By following the ideas of [70], carefully adapted to the neural setting, we show:

Theorem 4 (Approximate Counting). *For every time parameter t, and $\delta \in (0, 1)$, there exists a randomized construction of an approximate counting network using $O(\log\log t + \log(1/\delta))$ deterministic threshold gates plus an additional* single *random spiking neuron, that computes an $O(1)$ (multiplicative) approximation for the number of input spikes in t rounds with probability $1 - \delta$.*

We note that unlike the deterministic construction of timers that could be easily adapted to the problem of neural counting, our optimized randomized timers with $O(\log\log 1/\delta)$ neurons cannot be adopted into an approximate counter network. We therefore solve the latter by adopting Morris algorithm to the neural setting.

Broader scope: lessons from dynamic streaming algorithms. We believe that the approximate counting problem provides just one indication of the potential relation between succinct neural networks and dynamic streaming algorithms. In both settings, the goal is to gather statistics (e.g., over time) using a small amount of space. In the setting of neural networks, there are additional difficulties that do not show up in the streaming setting. E.g., it is also required to obtain fast *update time*, as illustrated in our solution to the approximate counting problem. See Chapter 3 for more details on the connection to streaming algorithms.

2.1.2 Neural Synchronizers

The standard model of spiking neural networks assumes that all edges (synapses) in the network have a uniform response latency. That is, the electrical signal is passed from the presynaptic neuron to the postsynaptic neuron within a fixed time unit which we call a *round*. However, in real biological networks, the response latency of synapses can vary considerably depending on the biological properties of the synapse, as well as on the distance between the neighboring neurons. This results in an asynchronous setting in which different edges have distinct response times. We formalize a simple model of spiking neurons in the asynchronous setting, in which the given neural network also specifies a *response latency* function $\ell : A \to \mathbb{R}_{\geq 1}$ that determines the number of rounds it takes for the signal to propagate over the edge. Inspired by the synchronizers of Awerbuch and Peleg [15], and using the above-mentioned compressed timer and counter modules, we present a general simulation methodology (a.k.a synchronizers) that takes a network $\mathcal{N}_{\mathsf{sync}}$ that solves the problem in the synchronized setting, and transform it into an “analogous” network $\mathcal{N}_{\mathsf{async}}$ that solves the same problem in the asynchronous setting.

The basic building block of this transformation is the neural timer component adapted to the asynchronous setting. The cost of the transformation is measured by the overhead in the number of neurons and in the computation time. Using our neural timers leads to a small overhead in the number of neurons.

Theorem 5 (Synchronizer, Informal)**.** *There exists a synchronizer that given a network $\mathcal{N}_{\mathsf{sync}}$ with n neurons and maximum response latency*[1] *L, constructs a network $\mathcal{N}_{\mathsf{async}}$ that has an "analogous" execution in the asynchronous setting with a total number of $O(n + L\log L)$ neurons and a time overhead of $O(L^3)$.*

We note that although the construction is inspired by the work of Awerbuch and Peleg [15], due to the large differences between these models, the precise formulation and implementation of our synchronizers are quite different. The most notable difference between the distributed and neural settings is the issue of memory: in the distributed setting, vertices can aggregate the incoming messages and respond when all required messages have arrived. In striking contrast, our neurons can only respond (by either firing or not firing) to signals arrived in the *previous* round, and all signals from previous rounds cannot be locally stored. For this reason and unlike [15], we must assume a bound on the largest edge latency. In particular, in [86] we show that the size overhead of the transformed network $\mathcal{N}_{\mathsf{async}}$ must depend, at least logarithmically, on the value of the largest latency L.

Observation 1. *The size overhead of any synchronization scheme is $\Omega(\log L)$.*

This provably illustrates the difference in the overhead of synchronization between general distributed networks and neural networks. We leave the problem of tightening this lower bound (or upper bound) as an interesting open problem.

Additional related work. To the best of our knowledge, there are two main previous theoretical works on asynchronous neural networks. Maass [126] considered a quite elaborated model for deterministic neural networks with *arbitrary* response *functions* for the edges, along with latencies that can be chosen by the network designer. Within this generalized framework, he presented a coarse description of a synchronization scheme that consists of various time modules (e.g., initiation and delay modules). Our work complements the scheme of [126] in the simplified SNN model by providing a rigorous implementation and analysis for size and time overhead. Khun et al. [108] analyzed the synchronous and asynchronous behavior under the stochastic neural network model of DeVille and Peskin [55]. Their model and framework are quite different from ours and do not aim at building synchronizers.

Turning to the setting of logical circuits, there is a long line of work on the asynchronous setting under various model assumptions [11, 82, 180, 28, 135] that do not quite fit the memoryless setting of spiking neurons.

2.2 Deterministic Constructions of Neural Timer Networks

In this section, we consider *deterministic* neural timer networks, defined as follows.

[1] I.e., L correspond to the *length* of the longest round.

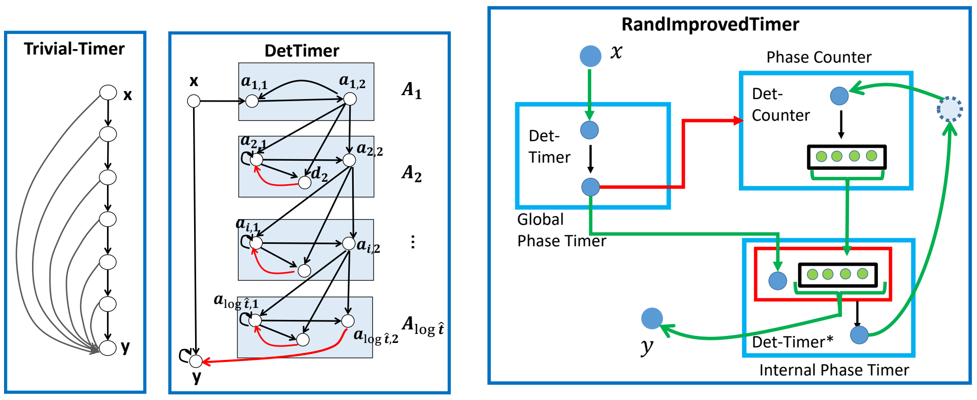

Figure 2.1: Illustration of timer networks with time parameter t. Left: The naïve timer with $\Theta(t)$ neurons. Mid: deterministic timer with $\Theta(\log t)$ neurons. Right: randomized timer with $O(\log\log 1/\delta)$) neurons, using the DetTimer modules with parameter $t' = \log 1/\delta$.

Definition 1 (Det. Neural Timer Network)**.** *Given time parameter t, a* deterministic *neural timer network $\mathcal{DT}$ is a network of threshold gates, with an input neuron x, an output neuron y, and additional auxiliary neurons. The network satisfies that in every round τ, $\sigma_\tau(y) = 1$ iff there exists a round $\tau > \tau' \geq \tau - t$ such that $\sigma_{\tau'}(x) = 1$.*

For a given neural timer network $\mathcal{N}$ with N auxiliary neurons, the *state* of the network in round τ denoted as σ_τ, is described by an N-length vector indicating the firing neurons in that round. We start by observing that neural networks have a memoryless property, in the sense that each state depends only on the state of the previous round.

Observation 2 (Memoryless Property)**.** *Given a deterministic neural network $\mathcal{N}$, for any round $\tau > 0$, the state σ_τ is fully determined by $\sigma_{\tau-1}$.*

Similarly, given an SNN network, for every fixed state s^* and round τ it holds that

$$\Pr[\sigma_\tau = s^* \mid \sigma_1, ...\sigma_{\tau-1}] = \Pr[\sigma_\tau = s^* \mid \sigma_{\tau-1}].$$

Lower bound (Poof of Theorem 1(2)). Assume towards contradiction that there exists a neural timer with $N \leq \log t - 1$ auxiliary neurons. Since there are at most 2^N different states, by the pigeonhole principle, there must be at least two rounds $\tau, \tau' \leq t - 1$ in which the state of the network is identical, i.e., where $\sigma_\tau = \sigma_{\tau'} = s^*$ for some $s^* \in \{0,1\}^N$. By the correctness of the network, the output neuron y fires in all rounds $\tau'' \in [\tau + 1, \tau' + 1]$. By the memoryless property (Obs. 2), we get that $\sigma_{\tau''} = s^*$ for $\tau'' = \tau + i \cdot (\tau' - \tau)$ for every $i \in \mathbb{N}_{\geq 0}$. Thus, y continues firing forever, in contradiction to the requirement that it stops firing after t rounds. Note that this lower bound holds even if y is allowed to stop firing in any finite time window.

A matching upper bound (Proof. Theorem 1(1)). For ease of explanation, we first sketch here the description of the network assuming that it is applied only once (i.e., the input x fires once within a window of t rounds). Taking care of the general case requires slight adaptations[1], Section 2.2.1 for the complete details. At the high-level, the network consists of $k = \Theta(\log t)$ layers $A_1, \ldots, A_k$ each containing two excitatory neurons $a_{i,1}, a_{i,2}$ denoted as *counting* neurons, and one inhibitory neuron d_i. Each layer A_i gets its input from layer A_{i-1} for every $i \geq 2$, and A_1 gets its input from x. The role of each layer A_i is to count *two* firing events of the neuron $a_{i-1,2} \in A_{i-1}$. Thus the neuron $a_{\log t,2}$ counts $2^{\log t}$ many rounds.

Because our network has an update time of $\log t$ rounds (i.e., number of rounds to update the timer), for a given time parameter t, the construction is based on the parameter $\hat{t}$ where $\hat{t} + \log \hat{t} = t$.

- The first layer A_1 consists of two neurons $a_{1,1}, a_{1,2}$. The first neuron $a_{1,1}$ has positive incoming edges from x and $a_{1,2}$ with weights $w(x, a_{1,1}) = 3$, $w(a_{1,2}, a_{1,1}) = 1$, and threshold $b(a_{1,1}) = 1$. The second neuron $a_{1,2}$ has an incoming edge from $a_{1,1}$ with weight $w(a_{1,1}, a_{1,2}) = 1$ and threshold $b(a_{1,2}) = 1$. Because we have a loop going from $a_{1,1}$ to $a_{1,2}$ and back, once x fired $a_{1,2}$ will fire every two rounds.

- For every $i = 2 \ldots \log \hat{t}$, the i-th layer A_i contains 3 neurons, two *counting* neurons $a_{i,1}$, $a_{i,2}$ and a reset neuron d_i. The first neuron $a_{i,1}$ has positive incoming edges from $a_{i-1,2}$, and a self-loop with weights $w(a_{i-1,2}, a_{i,1}) = w(a_{i,1}, a_{i,1}) = 1$, a negative incoming edge from d_i with weight $w(d_i, a_{i,1}) = -1$, and threshold $b(a_{i,1}) = 1$. The second counting neuron $a_{i,2}$ has incoming edges from $a_{i-1,2}$ and $a_{i,1}$ with weights $w(a_{i-1,2}, a_{i,2}) = w(a_{i,1}, a_{i,2}) = 1$, and threshold $b(a_{i,2}) = 2$. The reset neuron d_i is an inhibitory copy of $a_{i-1,2}$ and therefore also has incoming edges from $a_{i-1,2}$ and $a_{i,1}$ with weight $w(a_{i-1,2}, d_i) = w(a_{i,1}, d_i) = 1$ and threshold $b(d_i) = 2$. As a result, $a_{i,1}$ starts firing after $a_{i-1,2}$ fires once, and $a_{i,2}$ fires after $a_{i-1,2}$ fires twice. Then the neuron d_i inhibits $a_{i,1}$ and the layer is ready for a new count.

- The output neuron y has a positive incoming edge from x as well as a self-loop with weights $w(x, y) = 2$, $w(y, y) = 1$. In addition, it has a negative incoming edge from the last counting neuron $a_{\log \hat{t},2}$ with weight $w(a_{\log \hat{t},2}, y) = -1$ and threshold $b(y) = 1$. Hence, after x fires the output y continues to fire as long as $a_{\log \hat{t},2}$ did not fire.

- The (inhibitory) last counting neuron $a_{\log \hat{t},2}$ also has negative outgoing edges to all counting neurons (neurons of the form $a_{i,j}$) with weight $w(a_{\log \hat{t},2}, a_{i,j}) = -2$. As a result, after the timer counts t rounds it is reset.

The key claim that underlines the correctness of Theorem 1(1) is as follows.

Claim 1. *If x fires in round t_0, for each layer i the neuron $a_{i,2}$ fires in rounds $t_0 + \ell \cdot 2^i + i - 1$ for every $\ell = 1 \ldots \lfloor \hat{t}/2^i \rfloor$.*

[1] I.e., whenever x fires again in a window of t rounds, one should reset the timer and start counting t rounds from that point on.

Proof. The proof is by induction on i. For $i = 1$, once x fires in round t_0, neuron $a_{1,1}$ fires in round $t_0 + 1$ and $a_{1,2}$ fires in round $t_0 + 2$. Because there is a bidirectional edge between $a_{1,1}$ and $a_{1,2}$, the second counting neuron $a_{1,2}$ keeps firing every two rounds. Assume the claim holds for neuron $a_{i-1,2}$, and consider the i-th layer A_i. Recall that $a_{i,2}$ fires in round t' only if $a_{i,1}$ and $a_{i-1,2}$ fired in round $t' - 1$. The neuron $a_{i,1}$ fires one round after $a_{i-1,2}$ fires and keeps firing as long as d_i did not fire. By the induction assumption $a_{i-1,2}$ fired for the first time in round $2^{i-1} + i - 2$ and therefore $a_{i,1}$ starts firing in round $2^{i-1} + i - 1$. Note that in round $2^{i-1} + i - 1$ the neuron $a_{i-1,2}$ did not fire, and therefore the neurons $a_{i,2}$ and d_i can start firing only after $a_{i-1,2}$ fires again. Hence, only in round $2 \cdot 2^{i-1} + i - 2 + 1 = 2^i + i - 1$ the neurons $a_{i,2}$ and d_i fires for the first time. In the next round, because of the inhibition of d_i both counting neurons $a_{i,1}$ and $a_{i,2}$ do not fire and we can repeat the same arguments considering the next time the counting neurons $a_{i,1}$, $a_{i,2}$ fire.

We note that once the neuron $a_{\log \hat{t},2}$ fires for the first time in round $t_0 + 2^{\log \hat{t}} + \log \hat{t} - 1 = t_0 + \hat{t} + \log \hat{t} - 1$, it inhibits all the counting neurons. Hence, as long as x did not fire again, all counting neurons will be idle starting at round $t_0 + \hat{t} + \log \hat{t} = t_0 + t$. □

2.2.1 Adaptation of DetTimer to the General Case

We extend the DetTimer(t) network to handle the case where x fired more than once within the execution.

- **Case 1: x fires several times within a span of t rounds.** We introduce an additional reset (inhibitory) neuron r that receives input from x with weight $w(x, r) = 1$, and had threshold value $b(r) = 1$. The reset neuron r has outgoing edges to all neurons except $a_{1,2}$ and y with negative weight of -2

- **Case 2: x fires again just one round before $a_{\log \hat{t},2}$ fires.** To process this new spike, we introduce a control neuron c that receives input from x with weight $w(x, c) = 1$, has threshold $b(c) = 1$, and therefore fires one round after x. The control neuron c has outgoing edges to y and $a_{1,2}$ with weights $w(c, y) = w(c, a_{1,2}) = 3$. Therefore even if $a_{\log \hat{t},2}$ fires one round after x, the control neuron will cancel the inhibition on the output y and on $a_{1,2}$ and the timer will continue to fire.

Fig. 2.2 illustrates the structure of the network.

We next use Claim 1 in order to prove the first part of Theorem 1.

Complete Proof of Theorem 1(1)

Proof. We start by considering the case where x fires once in round t'. If x fired in round t', due to the self-loop of y, starting from round $t' + 1$, the output keeps firing as long as $a_{\log \hat{t},2}$ did not fire. By Claim 1, $a_{\log \hat{t},2}$ fires in round $t' + \hat{t} + \log \hat{t} - 1 = t' + t - 1$, and therefore y will be

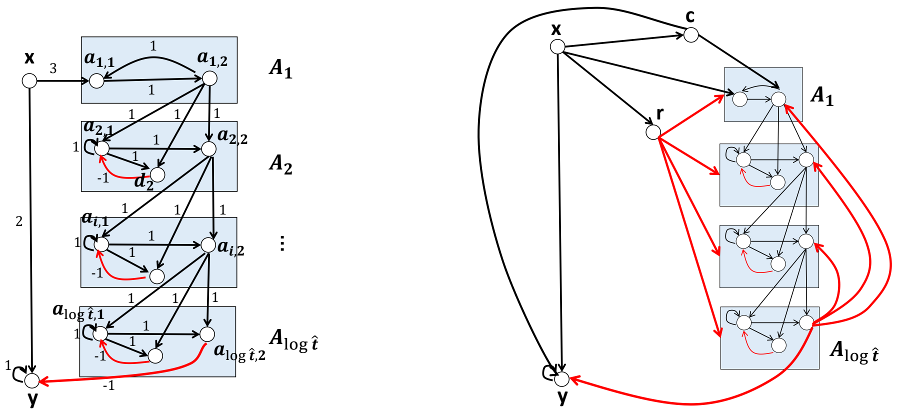

Figure 2.2: Illustration of the DetTimer network. Left: The simplified network for the case that x fired once. The neurons y, $a_{1,2}$ and the set of neurons $\{a_{1,1}, \ldots, a_{\log \hat{t},1}\}$ have threshold 1. For $i \geq 2$ the threshold of $a_{i,2}$ and d_i is 2. Right: A complete network description for the general case, where the input can fire several times during the execution. The reset neuron r resets the timer in case x fires several times. The control neuron c takes care of the special extreme case where x fires again one round *before* the last counting neuron $a_{\log \hat{t},2}$ fires.

inhibited in round $t' + t$. Note that $a_{\log \hat{t},2}$ also inhibits all other auxiliary neurons, and therefore as long as x will not fire again, y will also not fire. Next, we consider the case where x also fired in round $t'' \geq t' + 1$.

- **Case 1:** $t'' \geq t' + t$. Because in round $t' + t - 1$ the neuron $a_{\log \hat{t},2}$ inhibits all counting neurons in the network, starting round $t' + t$ no counting neuron fires until x fires again in round t''. Thus, after x fires in round t'', the network behaves the same as after the first firing event.

- **Case 2:** $t'' \leq t' + t - 3$. In round $t'' + 1 \leq t' + t - 2$ the reset neuron r inhibits all counting neurons except for $a_{1,2}$. Hence, in round $t'' + 2$ only y and $a_{1,2}$ fire, and the neural timer continues to count for additional $t - 2$ rounds.

- **Case 3:** $t'' = t' + t - 1$. The neuron $a_{\log \hat{t},2}$ fires on the same round as x. Since the weights on the edges from x to y and $a_{1,1}$ are greater than the weight of the inhibition from $a_{\log \hat{t},2}$, the timer continues to fire based on the last firing event of x.

- **Case 4:** $t'' = t' + t - 2$. In this case x fires in round t'' and in the next round, $a_{\log \hat{t},2}$ fires and inhibits the output y (at the same round that the reset neuron r fires). Recall that in round $t'' + 1$ the control neuron c also fires. Hence, in round $t'' + 2$ the neuron c excites y and $a_{1,2}$ canceling the inhibition of $a_{\log \hat{t},2}$.

□

2.2.2 Useful Modifications of Deterministic Timers

(1) Time parameter as a soft-wired input. We show a slightly modified variant of the neural timer denoted by $\mathsf{DetTimer}^*$ which receives as input an additional set of $\log t$ neurons that encode the desired duration of the timer. This modified variant is used in our improved randomized constructions.

Specifically, the $\mathsf{DetTimer}$ construction is modified to receive a time parameter $t' \leq t$ as a (soft) input to the network. That is, we assume that t is the upper limit on the time parameter. The same network can be used as a timer for any $t' \leq t$ rounds, and this t' can be given as an input to the network. In such a case, once the input neuron x fires, the output neuron y will fire for the next t' consecutive rounds. The time parameter t' is given in its binary form using $\log t$ input neurons denoted as $z_1 \ldots z_{\log t}$. We denote this network as $\mathsf{DetTimer}^*(t)$. The idea is that given a time parameter t', we want to use only $\log(t'')$ layers out of the $\log t$, where $t'' = t' + \log(t')$ (we use t'' due to the $\log(t'')$ delay in the update of the timer). The modifications are as follows.

1. The time input neurons $z_1, \ldots, z_{\log t}$ are set to be inhibitors.
2. We introduce an intermediate layer of neurons $c_1 \ldots c_{\log \hat{t}}$ that determines how many layers of counting neurons we should use. Each c_i has negative edges from $z_1, \ldots, z_{\log t}$ with weights $w(c_i, z_j) = -2^{j-1}$, and threshold $b(c_i) = -i - 1 - 2^{i-1}$. Hence c_i fires iff $i - 1 + 2^{i-1} \geq \mathsf{dec}(\bar{z}) = t'$.
3. We introduce $\log \hat{t}$ inhibitors $r_1, \ldots r_{\log \hat{t}}$ in order to inhibit the output y after we count to t' and reached layer t''. Each r_i has incoming edges from c_i and $a_{i,1}$, and fires as an AND gate. Hence, each r_i fires only when the timer count reach $2^{i-1} + i - 1$ and $i - 1 + 2^{i-1} \geq t'$.
4. The output neuron y receives negative incoming edges from the neurons $r_1 \ldots r_{\log \hat{t}}$ with weight $w(r_i, y) = -1$, and stops firing if at least one neuron r_i fired in the previous round.
5. Every neuron r_i also has negative outgoing edges to all counting neurons $a_{j,k}$ $k \in \{1, 2\}, j = 1 \ldots \log \hat{t}$ with weight $w(r_i, a_{j,k}) = -2$ in order to reset the timer when we finish counting to t'.

See Fig. 2.3 for an illustration of $\mathsf{DetTimer}^*(t)$ network.

(2) Extension to neural counting. We next show a modification of the timer into a neural counter network $\mathsf{DetCounter}$ that instead of counting the number of rounds, counts the number of input spikes in a time interval of t rounds.

Lemma 1. *Given time parameter t, there exists a deterministic* neural counter *network which has an input neuron x, a collection of $\log t$ output neurons represented by a vector $\bar{y}$, and $O(\log t)$ additional auxiliary neurons. In a time window of t rounds, for every round τ, if x fired r_τ times in the last τ rounds, the output $\bar{y}$ encodes r_τ by round $\tau + \log r_\tau + 1$.*

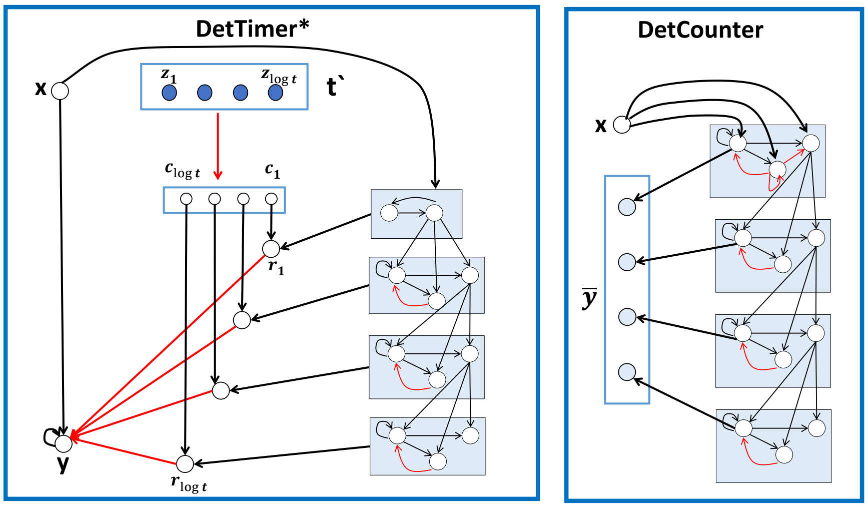

Figure 2.3: Left: Det. neural timer DetTimer* with a soft-wired time parameter. The input neurons $z_1, \ldots, z_{\log t}$ encode the time parameter t'. The intermediate neurons $c_1, \ldots, c_{\log t}$ control how many layers are used depending on the time parameter t'. Once the timer reaches layer $i = \Theta(\log(t'))$ for which c_i fires, the inhibitor r_i inhibits the output y and the counting terminates. Right: neural counter DetCounter, the output neurons $\bar{y}$ encode the number of times x fired in a time window of t rounds.

This extra-additive factor of $\log r_\tau$ is due to the update time of the counter. In Section 2.5, we revisit the neural counter problem and provide an *approximate* randomized solution with $O(\log\log t + \log(1/\delta))$ many neurons where δ is the error parameter. This construction is based on the well-known Morris algorithm (using the analysis of [70]) for approximate counting in the streaming model.

We next describe the required adaptations for constructing the network DetCounter. The DetCounter network with parameter t contains $\log t$ layers, all layers $i \geq 2$ are the same as in DetTimer and only the first layer is slightly modified. The first counting neuron $a_{1,1}$ has a positive incoming edge from x with weight $w(x, a_{1,1}) = 4$, and a self-loop with weight $w(a_{1,1}, a_{1,1}) = 1$. In addition $a_{1,1}$ has a negative edge from the inhibitor d_1 with weight $w(d_1, a_{1,1}) = -1$, and threshold $b(a_{1,1}) = 1$. The second counting neuron $a_{1,2}$ has positive edges from x and $a_{1,1}$ with weights $w(x, a_{1,2}) = w(a_{1,1}, a_{1,2}) = 1$, a negative edge from d_1 with weight $w(d_1, a_{1,2}) = -2$ and threshold $b(a_{1,2}) = 2$. The reset neuron d_1 is an inhibitory copy of $a_{1,2}$ and therefore also has positive edges from x and $a_{1,1}$ with weights $w(x, d_1) = w(a_{1,1}, d_1) = 1$, a negative self-loop with weight $w(d_1, d_1) = -2$ and threshold $b(d_1) = 2$. We then connect the counting neurons $a_{1,1}, \cdots a_{\log t,1}$ to the output vector directly, where y_i has an incoming edge from $a_{i,1}$ with weight $w(a_{i,1}, y_i) = 1$ and threshold $b(y_i) = 1$. Fig. 2.3 demonstrates the DetCounter(t) network.

Next, we show that once the counter is updated, the number of times that x fired is represented as a binary number where the counting neuron $a_{i,1}$ represents the i-th bit in the binary representation ($a_{1,1}$ is the least significant bit). We note that if the last firing of x occurs in round

τ then after at most $\log c + 1$ rounds the counter is updated with the new value, where c is the value of the counter before round τ. We start with the following claim concerning the first layer.

Claim 2. *If x fired in round τ, the neurons d_1 and $a_{1,2}$ fire in round $\tau + 1$ iff x fired an even number of times by round τ.*

Proof. By induction on the number of times x fired, denoted as n. Since d_1 and $a_{1,2}$ have identical potential functions it is sufficient to prove the claim for the neuron d_1. For $n = 1$, if x fired once in round τ, then $a_{1,1}$ fires for the first time in round $\tau + 1$, and since d_1 fires only if $a_{1,1}$ fired in the previous round, in round $\tau + 1$ both neuron d_1 and $a_{1,2}$ are idle. For $n = 2$, since x fired for the first time in some round $\tau' \leq \tau - 1$, starting round $\tau' + 1$ neuron $a_{1,1}$ fires on every round until d_1 fires. Hence, in round $\tau + 1$ the neuron d_1 receives spikes from both x and $a_{1,1}$ and therefore fires. Assume the claim holds for every $k \leq n - 1$ and we will show correctness for n. Denote the round in which x fired for the $(n - 1)$-th time by $\tau' \leq \tau - 1$.

- (Case 1: n is even.) Since $n - 1$ is odd, by the induction assumption d_1 did not fire in round $\tau' + 1$. Hence $a_{1,1}$ is not inhibited until round $\tau + 1$, and due to the self-loop $a_{1,1}$ also fires in round τ. Therefore d_1 and $a_{1,2}$ fire in round $\tau + 1$.
- (Case 2: n is odd.) If $\tau' = \tau - 1$, by the induction assumption d_1 fires in round $\tau' + 1 = \tau$, and due to the negative edges from d_1, both d_1 and $a_{1,2}$ are idle in round $\tau + 1$. Otherwise, $\tau' \leq \tau - 2$. By the induction assumption, d_1 fires in round $\tau' + 1$. Since x did not fire in round $\tau' + 1$ (as it fires again only in round τ), in round $\tau' + 2 \leq \tau$ the neuron $a_{1,1}$ is inhibited by d_1 and therefore in round τ the neurons d_1 and $a_{1,2}$ receives a signal only from x and does not fire.

□

Next, we show that if x fired in round τ for the last time, for each layer $i \in [1, \log n]$, neuron $a_{i,2}$ fires in round $\tau + i$ only if x fired $\ell \cdot 2^{i-1}$ times by round τ for some integer $\ell \geq 1$.

Claim 3. *For every layer $i \in [2, \log t]$, if $a_{i-1,2}$ fired in round τ for the n-th time, the neurons d_i and $a_{i,2}$ fire in round $\tau + 1$ iff n is even.*

Proof. By induction on n. For $n = 1$, one round after the first time neuron $a_{i-1,2}$ fires, the neuron $a_{i,1}$ fires for the first time, and therefore $a_{i,2}$, d_i do not fire. For $n = 2$, the second time $a_{i-1,2}$ fires, due to the self-loop on $a_{i,1}$ it fires as well, and therefore after one round $a_{i,2}$ and d_i fire. Assume that $a_{i-1,2}$ fired in round τ' for the $(n - 1)$-th time. If n is even, then by the induction assumption d_i does not fire in round $\tau' + 1 \leq \tau$. Hence, due to the self-loop of $a_{i,1}$, in round τ also $a_{i,1}$ fires and therefore d_i and $a_{i,2}$ fire in round $\tau + 1$. If n is odd, by the induction assumption d_i fires in round $\tau' + 1$. By Claim 2, there is at least one round distance between every two firing events of $a_{1,2}$. Thus, there is at least one round distance between every two firing events of $a_{i-1,2}$, and therefore $\tau \geq \tau' + 2$. Hence, because $a_{i,1}$ was inhibited by d_i in round $\tau' + 1 < \tau$, it is idle in round τ and the neurons d_i and $a_{i,2}$ do not fire in round $\tau + 1$. □

Corollary 1. *If x fired for the n-th time in round τ, for every layer $i \in [1, \log t]$ the neurons d_i and $a_{i,2}$ fire in round $\tau + i$ iff $(n \bmod 2^i) = 0$.*

Proof. By induction on i. The base case for $i = 1$ follows from Claim 2. Assume that the claim holds for layer i and we will show it also holds for layer $i + 1$. If $(n \bmod 2^i) = 0$, then $n = q \cdot 2 \cdot 2^{i-1}$ for some integer q. Therefore by the induction assumption, $a_{i,2}$ fires in round $\tau + i$, and moreover it fired an even number of times by that round. Hence, by Claim 3 the neurons d_{i+1} and $a_{i+1,2}$ fire in round $\tau + i + 1$. Otherwise, if $(n \bmod 2^i) \neq 0$, by the induction assumption $a_{i,2}$ does not fire in round $\tau + i$ and therefore d_{i+1} and $a_{i+1,2}$ do not fire in round $\tau + i + 1$. If $(n \bmod 2^i) = 0$ but $(n \bmod 2^{i+1}) \neq 0$, then by the induction assumption $a_{i,2}$ fired an odd number of times by round $\tau + i$ and by Claim 3 neurons d_i and $a_{i,2}$ do not fire in round $\tau + i + 1$. □

Proof of Lemma 1. The first counting neuron $a_{i,1}$ fires one round after $a_{i-1,2}$ fires, and as long as d_i and $a_{i,2}$ did not fire. Hence, by Cor. 1 we can conclude that if x fired for the last time in round τ, by round $\tau + \log r_\tau + 1$, the neurons $a_{1,1}, \ldots, a_{\log t,1}$ hold a binary representation of the number of times r_τ that x fired by round τ.

2.3 Randomized Constructions of Neural Timer Networks

We now turn to consider randomized implementations. The input to the construction are a time parameter t and an error probability $\delta \in (0, 1)$, that are hard-wired into the network.

Definition 2 (Rand. Neural Timer Network)**.** *A randomized neural timer $\mathcal{RT}$ for parameters $t \in \mathbb{N}_{>0}$ and $\delta \in (0, 1)$, satisfies the following for a time window of $\mathsf{poly}(t)$ rounds.*

- *For every fixed firing event of x in round τ, with probability $1 - \delta$, y fires in each of the following t rounds.*
- *$\sigma_{\tau'}(y) = 0$ for every round τ' such that $\tau' - \mathsf{Last}(\tau') \geq 2f$ with probability $1 - \delta$, where $\mathsf{Last}(\tau') = \max\{i \leq \tau' \mid \sigma_i(x) = 1\}$ is the last round τ in which x fired up to round τ'.*

Note that in our definition, we have a success guarantee of $1 - \delta$ for any fixed firing event of x, on the event that y fires for t many rounds after this firing. In contrast, with a probability of $1 - \delta$ over the entire span of $\mathsf{poly}(t)$ rounds, y *does not* fire in cases where the last firing of x was $2f$ rounds apart. We start by showing a simple construction with $O(\log 1/\delta)$ neurons.

2.3.1 Warm Up: Randomized Timer with $O(\log 1/\delta)$ Neurons

The network $\mathsf{BasicRandTimer}(t, \delta)$ contains a collection of $\ell = \Theta(\log 1/\delta)$ spiking neurons $A = \{a_1, \ldots, a_\ell\}$ that can be viewed as a *time-estimator population*. Each of these neurons has a positive self-loop, a positive incoming edge from the input neuron x, and a positive outgoing edge to the output neuron y. See Fig. 2.4 for an illustration. Whereas these a_i neurons are probabilistic spiking neurons[1], the output y is simply a threshold gate. We next explain the underlying intuition. Assume that the input x fired in round 0. It is then required for the output neuron y to fire for at least t rounds $1, \ldots, t$, and stop firing after at most $2f$ rounds with probability $1 - \delta$. By having every neuron a_i fire (independently) w.p $(1 - 1/t)$ in each round given that it fired in the previous round[2], we get that a_i fires for t *consecutive* rounds with probability $(1 - 1/t)^t \approx 1/e$. On the other hand, it fires for $2f$ consecutive rounds with probability $(1 - 1/t)^{2f} = 1/e^2$. Since we have $\Theta(\log 1/\delta)$ many neurons, by a simple application of Chernoff bound, the output neuron y (which simply counts the number of firing neurons in A) can distinguish between round t and round $2f$ with probability $1 - \delta$.

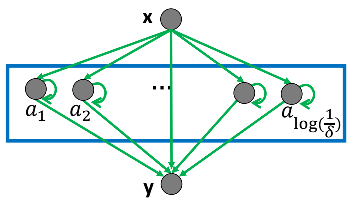

Figure 2.4: Illustration of the $\mathsf{BasicRandTimer}(t, \delta)$ network. Each neuron a_i fires with probability $1 - 1/t$ in round τ given that it fired in the previous round, and therefore fires for t consecutive rounds with constant probability. The output y fires if at least $1/(2e)$ fraction of the a_i neurons fired in the previous round.

Detailed construction. The network $\mathsf{BasicRandTimer}(t, \delta)$ has input neuron x, output neuron y, and $\ell = \Theta(\log 1/\delta)$ spiking neurons $A = \{a_1, \ldots, a_\ell\}$. We set the weights of the self-loop of each a_i, and the weight of the incoming edge from x to be $w(x, a_i) = w(a_i, a_i) = \log(t - 1) + b(a_i)$. The threshold value of a_i is set to $b(a_i) = \Theta(\log(t\ell/\delta))$. This makes sure that given a firing of either x or a_i in round τ, the probability that a_i fires in round $\tau + 1$ is $1 - 1/t$. In the complementary case (neither x nor a_i fired in round τ), a_i fires in round τ with probability at most $O(\delta/\mathsf{poly}(t\ell))$. For the output y, we set $w(a_i, y) = 1$ for each a_i, the weight of the edge from x to be $w(x, y) = \frac{\ell}{2e}$, and its threshold $b(y) = \frac{\ell}{2e}$. This makes sure that y fires in round τ' if either x or at least $1/2e$ fraction of the a_i neurons fired in round $\tau' - 1$. We next analyze the construction.

Lemma 2 (Correctness). *Within a time window of* $\mathsf{poly}(t)$ *rounds it holds that:*

[1] A neuron that fires with a probability that depends on its potential as specified in Chapter 1.

[2] A neuron a_i that stops firing in a given round, drops out and would not fire again with good probability.

- *For every fixed firing event of x in round τ, with probability $1-\delta$, y fires in each of the following t rounds.*

- $\sigma_{\tau'}(y) = 0$ *for every round τ' such that $\tau' - \mathsf{Last}(\tau') \geq 2f$ with probability at least* $1-\delta$.

Proof. When x fires in round τ_0, each neuron a_i fires for the following t consecutive rounds independently with probability $1/e$. Therefore, the expected number of neurons in A that fired for t consecutive rounds starting round τ_0+1 is $\frac{\ell}{e}$. Using Chernoff bound upon picking a large enough constant c s.t $\ell = c \cdot \log(1/\delta)$, at least $\ell/2e$ auxiliary neurons fired for t consecutive rounds and y fires in rounds $[\tau_0+2, \tau_0+t]$ with probability $1-\delta$. Since y has an incoming edge from x, it fires in round τ_0+1 as well.

Next, recall that for each neuron $a_i \in A$, given that a_i or x did not fire in round τ, the probability that a_i fires in round $\tau+1$ is at most $\delta/\mathsf{poly}(\ell t)$. Hence by union bound, in a window of $\mathsf{poly}(t)$ rounds, the probability there exists a neuron $a_i \in A$ that fired in round τ' but did not fire in round $\tau'-1$ is at most $\delta/2$. Assuming no $a_i \in A$ fires unless it fired previously, each $a_i \in A$ fires for $2f$ consecutive rounds with probability $1/e^2$. Using Chernoff bound the probability at least $\frac{\ell}{2e}$ neurons from A fired for $2f$ consecutive rounds is at most $\delta/2$ (again we choose ℓ accordingly). Thus, we conclude that the probability there exists a round τ' s.t $\tau' - \mathsf{Last}(\tau') \geq 2f$ in which $\sigma_{\tau'}(y) = 1$ is at most δ. □

2.3.2 Improved Construction with $O(\log\log 1/\delta)$ Neurons

We next describe an optimal randomized timer $\mathsf{ImprovedRandTimer}$ with an exponentially improved number of auxiliary neurons. This construction also enjoys the fact that it requires a *single* spiking neuron, while the remaining neurons can be deterministic threshold gates. Due to the tightness of Chernoff bound, one cannot really hope to estimate time with probability $1-\delta$ using $o(\log(1/\delta))$ samples. Our key idea here is to generate the same number of samples by re-sampling one particular neuron over several rounds. Intuitively, we are going to show that for our purposes having $\ell = \log(1/\delta)$ neurons $a_1, \ldots, a_\ell$ firing with probability $1-1/t$ in a *given* round is *equivalent* to having a *single* neuron a^* firing with probability $1-1/t$ (independently) in a sequence of ℓ rounds.

Specifically, observe that the distinction between round t and $2f$ in the $\mathsf{BasicRandTimer}$ network is based only on the *number* of spiking neurons in a given round. In addition, the distribution on the number of times a^* fires in a span of ℓ rounds is equivalent to the distribution on the number of firing neurons $a_1, \ldots, a_\ell$ in a given round. For this reason, every phase of $\mathsf{ImprovedRandTimer}$ simulates a single round of $\mathsf{BasicRandTimer}$. To count the number of firing events in ℓ rounds, we use the deterministic neural counter module with $\log \ell = O(\log\log 1/\delta)$ neurons.

We now further formalize this intuition. The network ImprovedRandTimer simulates each round of BasicRandTimer using a phase of $\ell' = \Theta(\log 1/\delta)$ rounds [1], but with only $O(\log\log 1/\delta)$ neurons. In the BasicRandTimer network each of the neurons a_i fires (independently) in each round w.p $1 - 1/t$. Once it stops firing in a given round, it basically drops out and would not fire again with good probability. Formally, consider an execution of the BasicRandTimer and let n_i be the number of neurons in A that fired in round i. In round $i+1$ of this execution, we have n_i many neurons each firing w.p $1 - 1/t$ (while the remaining neurons in A fire with a very small probability). In the corresponding $i+1$ phase of the network ImprovedRandTimer, the chief neuron a^* fires w.p $1 - 1/t'$ where $t' = \frac{t}{\ell'}$ for $n'_i \leq \ell$ consecutive rounds[2] where n'_i is the number of rounds in which a^* fired in phase i.

The dynamics of the network ImprovedRandTimer is based on discrete phases. Each phase consists of a fixed number of $\ell' = O(\ell)$ rounds but has a possibly different number of *active rounds*, namely, rounds in which a^* attempts firing. Specifically, a phase i has an active part of n'_i rounds where n'_i is the number of rounds in which a^* fired in phase $i-1$. In the remaining $\ell' - n'_i$ rounds of that phase, a^* is idle. To implement this behavior, the network should keep track of the number of rounds in which a^* fires in each phase, and supply it as an input to the next phase (as it determines the length of the active part of that phase). For that purpose, we will use the deterministic modules of neural timers and counters. The module DetCounter with time parameter $\Theta(\log 1/\delta)$ is responsible for counting the number of rounds that a^* fires in a given phase i. The output of this module at the end of the phase is the input to a DetTimer* module[3] at the beginning of phase $i+1$. In addition, we also need a *phase timer* module DetTimer with time parameter $\Theta(\log 1/\delta)$ that "announces" the end of a phase and the beginning of a new one. Similarly to the network BasicRandTimer, the output neuron y fires as long as a^* fires for at least $(1/2e)$ fraction of the rounds in each phase (in an analogous manner as in the BasicRandTimer construction). See Fig. 2.5 for an illustration of the network. Note that since we only use deterministic modules with time parameter $\Theta(\log 1/\delta)$, the total number of neurons (which are all threshold gates) will be bounded by $O(\log\log 1/\delta)$. We next give a detailed description of the network and prove Theorem 2.

Complete proof of Theorem 2: We first describe the modules of the network ImprovedRandTimer which receives as input the time parameter t and error probability δ.

Network modules:

- A *Global-Phase-Timer* module implemented by a (slightly modified) module of DetTimer(ℓ'). Due to the update time of DetCounter (Lemma 1), we set the length of each phase to $\ell' = \ell + \log \ell$ where ℓ correspond to the number of spiking neurons in BasicRandTimer. Upon initializing this timer, the output neuron of this module fires *after* ℓ' rounds (instead

[1] Due to technical reasons each phase consists of $\ell' = \ell + \log \ell$ rounds instead of ℓ.

[2] Note that because each phase takes $\ell' = \Theta(\log 1/\delta)$ rounds, we will need to count $t' = \frac{t}{\ell'}$ many phases. Thus a^* fires with probability $1 - 1/t'$ rather than w.p $1 - 1/t$.

[3] Here we use the variant of DetTimer in which the time is encoded in the input layer of the network.

of firing *for* ℓ' rounds). This firing is the wake-up call for the network that a phase has terminated (ℓ' rounds have passed). This will activate some cleanup steps and a subsequent "announcement" for the start of a new phase.

To allow this module to inhibit as well as excite other neurons in the network, we will have two output (copy) neurons, one will be an inhibitory neuron and the other is an excitatory neuron. The inhibitor activates a clean-up round (in order to clear the counting information from the previous phase). After one round, using a delay neuron the excitatory neuron safely announces the beginning of a new phase.

- An *Internal-Phase-Timer* module also implemented by a (yet a differently slightly modified) variant of DetTimer. The role of this module is to indicate to the spiking neuron a^* the number of rounds in which it should attempt firing in each phase. Recall that each phase i starts by an active part of length n_i in which a^* attempts firing w.p. $1 - 1/t'$ in each of these rounds. In the remaining $\ell' - n_i$ rounds till the end of the phase, a^* is idle. In each phase i, we then set the internal timer to n_i, this will activate a^* for n_i rounds. The time parameter n_i is given as input to this module. For that purpose, we use the DetTimer* variant in which the time parameter is given as an input. In our case, this input is supplied by the output layer of the counting module (describe next) at the end of phase $i - 1$. In particular, at the end of the phase, the output of the counting module is fed into the input layer of the *Internal-Phase-Timer* module. Then, the information will be deleted from the counting module, ready to maintain the counting in the next phase.

  Since we would need to keep on providing the counting information throughout the entire phase, we augment the input layer of this module by self-loops that keep on presenting this information throughout the phase.

- A *Phase-Counter* module implemented by the DetCounter network, maintains the number of rounds in which a^* fires in the current phase. At the end of every i-th phase, the output layer of this module stores the number of rounds in which a^* fired in phase i. At the end of the phase, upon receiving a signal from the *Global-Phase-Timer*, the output layer copies its information to the input layer of the *Internal-Phase-Timer* module using an intermediate layer of neurons, and the information of the module is deleted (by inhibitory connections from the *Global-Phase-Timer* module).

Complete network description.

- The neuron a^* has threshold $b(a^*) = \Theta(\log(\ell t/\delta))$, and a positive incoming edge from the output z_1 of the *Internal-Phase-Timer* module with weight $w(z_1, a^*) = \ln(t' - 1) + b(a^*)$. Therefore a^* fires with probability $1 - 1/t'$ if z_1 fired in the previous round, and w.h.p[1] does not fire otherwise.

[1]Here high probability refers to probability of $1 - \delta/\mathsf{poly}(t)$.

- Each neuron in the output of the *Phase-Counter* has a positive outgoing edge to an intermediate *copy* neuron c_i with weight 1. In addition, each c_i has a positive incoming edge from the *Global-Phase-Timer* excitatory output with weight 1, and threshold $b(c_i) = 2$. The copy neurons have outgoing edges to the input of the *Internal-Phase-Timer* and are used to copy the current count for the next phase.

- The inhibitory output of the *Global-Phase-Timer* has outgoing edges to all neurons in the *Internal-Phase-Timer* and *Phase-Counter* with weight -5. This is used to clean up the outdated counting information at the end of the phase.

- The excitatory output of the *Global-Phase-Timer* has an outgoing edge to a delay neuron d with weight 1 and threshold $b(d) = 1$. Hence, d fires one round after a phase ended, and alerts the beginning of the new phase. The neuron d has outgoing edges to the input of *Global-Phase-Timer* and *Internal-Phase-Timer* with large weight.

- The output neuron y has incoming edges from the time input neurons $q_1, \ldots, q_{\log \ell}$ of the *Internal-Phase-Timer* module each with weight $w(q_i, y) = 1$ and threshold $b(y) = \frac{\ell}{2e}$. Therefore y fires if a^* fired for at least $\frac{\ell}{2e}$ times in the previous phase. In addition, y has positive incoming edges from x and the delay neuron d of the *Global-Phase-Timer* module, each with weight $\frac{\ell}{2e}$. This insures that y also fires between phases.

- The *Global-Phase-Timer* input has an incoming edge from x with a large weight, in order to initialize the timer when the input x fires. In addition, x has outgoing edges with a large weight to the time input of the *Internal-Phase-Timer*, such that the decimal value of the input is set to ℓ.

All neurons except for a^* are threshold gates, see Fig. 2.5 for a schematic description of the ImprovedRandTimer network.

Correctness. For simplicity we begin by showing the correctness of the construction assuming that there is a single firing of the input x during a period of $2f$ rounds. Taking care of the general case requires minor modifications that are described at the end of this section.

Our goal is to show that each *phase* of the ImprovedRandTimer network is equivalent to a *round* in the BasicRandTimer network. Toward that goal, we start by showing that the length of the active part of phase i has the same distribution as the number of neurons n_{i-1} that fire in round $i-1$ in BasicRandTimer(t', δ), where $t' = t/\ell'$. In the BasicRandTimer(t', δ) construction, let $\bar{B}_i$ be a random variable indicating the event that there exists a neuron $a \in A$ which fired in round $\tau \leq i$ but a as well as x did not fire in round $\tau - 1$. Similarly, for the ImprovedRandTimer(t, δ) construction, let $\bar{B}'_i$ be a random variable indicating the event that there exists a phase $\tau \leq i$, where neuron a^* fired in an inactive round of phase τ. Note that in both constructions, the probability that a^* fired in an inactive round, and the probability that $a \in A$ fired given that it did not fire in the previous round is identical. Moreover, within a

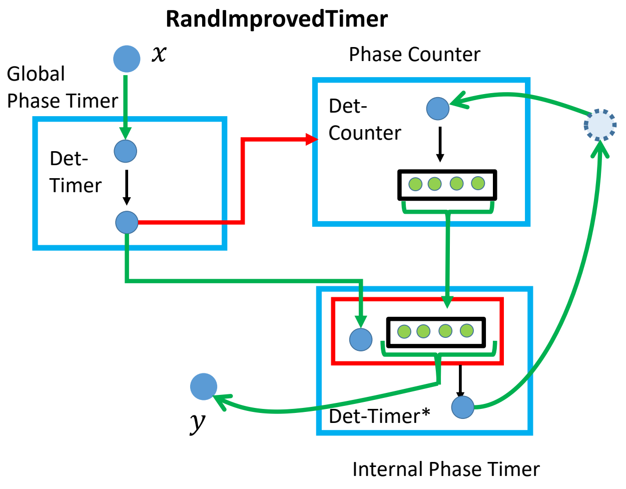

Figure 2.5: Schematic description of the randomized timer. In each module, only the input layer and the output layer are shown. Excitatory (inhibitory) relations are shown in green (red) arrows. Each module is deterministic and has $\Theta(\log\log 1/\delta)$ threshold gates. The lower right module (*Internal-Phase-Timer*) uses the variant of the deterministic neural timer in which the time parameter is softly encoded in the input layer. This is crucial as the length of the $(i+1)$-th active phase depends on the spike counts of a^* in phase i. This value is encoded by the output layer of the *Phase-Counter* module at the end of phase i. In contrast, the *Global-Phase-Timer* module uses the standard neural timer network (hard-wired), as the length of each phase is fixed.

window of $\tau = \text{poly}(t)$ rounds, by union bound both probabilities $\Pr[B_\tau]$ and $\Pr[B'_\tau]$ are at most $\delta/2$.

Let Y_i be a random variable for the number of neurons that fired in the i-th round in $\mathsf{BasicRandTimer}(t', \delta)$, and let X_i be the random variable for the number of rounds a^* fired during phase i in $\mathsf{ImprovedRandTimer}(t, \delta)$. In both constructions, we assume that the input neuron x fired only in round 0.

Claim 4. *For any $k \geq 0$ and $i \geq 1$,* $\Pr[X_i = k \mid \bar{B'}_i] = \Pr[Y_i = k \mid \bar{B}_i]$.

Proof. By induction on i. For $i = 1$, given $\bar{B}_1$, $\bar{B'}_1$, the random variable X_1 as well as Y_1 are the sum of ℓ independent Bernoulli variables with probability $1 - 1/t'$ and therefore $X_1 = Y_1$. Assume $\Pr[X_i = k \mid \bar{B}_i] = \Pr[Y_i = k \mid \bar{B}_i]$ and we will show the equivalence for $i+1$. First recall that in the $\mathsf{BasicRandTimer}(t', \delta)$ construction, each $a \in A$ fires with probability $1 - 1/t'$ given that it fired in the previous round. Moreover, conditioning on $\bar{B}_{i+1}$, given that a did not fire in round i, it does not fire in round $i+1$ as well. Thus, for any k, j it holds that $\Pr[Y_{i+1} = k \mid Y_i = j, \bar{B}_{i+1}] = \binom{j}{k}(1 - 1/t')^k \cdot (1/t')^{j-k}$ (i.e., a binomial distribution). Similarly, in the $\mathsf{ImprovedRandTimer}(t, \delta)$ construction, since we assumed that a^* fires only in the active rounds of each phase, given that a^* fired j times in phase i, in phase $i+1$ it holds that $\Pr[X_{i+1} = k \mid X_i = j, \bar{B'}_{i+1}] = \binom{j}{k}(1 - 1/t')^k \cdot (1/t')^{j-k}$. By the law of total probability, we conclude that

$$\begin{aligned}
\Pr[X_{i+1}=k \mid \bar{B'}_{i+1}] &= \sum_{j=0}^{\ell} \Pr[X_{i+1}=k \mid X_i=j, \bar{B'}_{i+1}] \cdot \Pr[X_i=j \mid \bar{B'}_{i+1}] \\
&= \sum_{j=0}^{\ell} \Pr[X_{i+1}=k \mid X_i=j, \bar{B'}_{i+1}] \cdot \Pr[Y_i=j \mid \bar{B}_i] \\
&= \sum_{j=0}^{\ell} \binom{j}{k} (1-1/t')^k \cdot (1/t')^{j-k} \cdot \Pr[Y_i=j \mid \bar{B}_i] \\
&= \sum_{j=0}^{\ell} \Pr[Y_{i+1}=k \mid Y_i=j, \bar{B}_{i+1}] \cdot \Pr[Y_i=j \mid \bar{B}_i] = \Pr[Y_{i+1}=k \mid \bar{B}_{i+1}],
\end{aligned}$$

where the second equality is due to the induction assumption. □

Hence, by the correctness of the network BasicRandTimer(t',δ), with probability at least $1-\delta$ the neuron a^* fires at least $\ell/2e$ times in each of the first t' phases. Since every phase consists of $\ell' = \ell + \log \ell$ rounds, y fires for at least $\ell' \cdot t' = t$ rounds w.h.p. On the other hand, with probability at most $\delta/2$ the neuron a^* fires in an inactive round during one of the first $2t'$ phases. Given that a^* fired only in active-rounds, we conclude that with probability at most $\delta/2$ the output y fires for at least $2f'$ phases. Altogether, with probability at least $1-\delta$ the output y stops firing by round $2f' \cdot \ell' = 2f$.

Finally, we describe the small modifications needed to handle the case where x fires several times within a window of $2f$ rounds. Upon any firing of x, all modules get reset, and a new counting starts. To implement the reset, we connect the input neuron x to two additional neurons, an inhibitory neuron x_1, and an excitatory neuron x_2 where $w(x,x_1) = w(x,x_2) = 1$ with thresholds $b(x_1) = b(x_2) = 1$. The inhibitor x_1 has outgoing edges to all auxiliary neurons in the network with weight -4. The excitatory neuron x_2 has outgoing edges to the input of *Global-Phase-Timer* and the time input of the *Internal-Phase-Timer*, such that the decimal value is equal to ℓ with weights 6. Thus, after one round of cleaning-up, the network starts to account the last firing of x. The output y has incoming edges from x and x_2 each with weight $w(x,y) = w(x_1,y) = \frac{\ell}{2e}$, this makes y fire during the reset period.

2.3.3 A Matching Lower Bound

We now turn to show a matching lower bound with randomized spiking neurons. Assume towards contradiction there exists a randomized neural timer $\mathcal{N}$ for a given time parameter t with $N = o(\log\log 1/\delta)$ neurons that succeed with probability at least $1-\delta$. This implies that once x fired, y fires for t consecutive rounds with probability $1-\delta$. Moreover, there exists some constant $c \geq 2$ such that y stops firing after $(c-1)\cdot t$ rounds w.p $1-\delta$. Throughout the proof, we assume w.l.o.g. that x fired in round 0. Recall that the state of the network in time τ

denoted as s_τ can be described as an N-length binary vector. Since we have N many neurons, the number of distinct states (or configurations) is bounded by $S = 2^N = o(\log 1/\delta)$. We start by establishing useful auxiliary claims.

We first claim that because we have a relatively small number of states and the memoryless property of Obs. 2, in every window of t rounds there exists a state that occurs at least twice (and with sufficient distance). Let s^* be a state for which the probability that there exist rounds $t', t'' \leq t$ such that $\frac{t}{3 \cdot S} \leq t' - t'' \leq t$ and $s_{t'} = s_{t''} = s^*$ is at least $1/S$.

Claim 5. *There exists such a state s^*.*

Proof. Note that because $N = o(\log\log 1/\delta)$ and $1/\delta \leq 2^{\mathsf{poly}(t)}$ it holds that $\frac{t}{3 \cdot S} \geq 1$. We partition the interval $[0, t]$ into $2 \cdot (S + 1)$ balanced intervals, each of size $t/2(S + 1)$. Because we have only S different states, in every execution of the network for t many rounds, there must be a state that occurs in at least two even intervals. Thus, there exists a state s^* for which the probability that s^* occurred in two even intervals is at least $1/S$. Because each interval is of size $t/2(S + 1) \geq t/3S$ we conclude that the claim holds. $\square$

Next, we use the assumption that with probability at least $1 - \delta$ the output y fires in rounds $[0, t]$ combined with the memoryless property (Obs. 2) to show that given that state s^* occurred in round t', with a sufficiently large probability, s^* occurs again with a long enough interval, and y fires in all rounds between the two occurrences of s^*. Let $p(t') = \Pr[\exists t'' \in [t' + t/(3S), t' + t], s_{t''} = s^* \text{ and } \sigma_{t^*}(y) = 1 \ \forall t^* \in [t', t''] \mid s_{t'} = s^*]$. By Obs. 2, we have:

Observation 3. $p(t') = p(t'')$ *for every* t', t''.

Define $p^* = p(1) = p(t')$ for any round t'. The next claim shows that p^* is sufficiently large.

Claim 6. $p^* \geq 1/S - \delta$.

Proof. Let A be an indicator random variable for the event that there exist $0 < t', t'' < t$ such that $t'' - t' \in [t/3S, t]$, $s_{t'} = s_{t''} = s^*$. Let B be the indicator random variable for the event that there exists $t^* \in [0, t]$ such that $\sigma_{t^*}(y) = 0$. By Claim 5, $\Pr[A] \geq 1/S$, and by the success guarantee of the network, $\Pr[B] \leq \delta$. Hence, by union bound, we get $\Pr[A \wedge \bar{B}] \geq 1/S - \delta$.

Let $A(t', t'')$ be the indicator random variable for the event that $s_{t'} = s_{t''} = s^*$ and $\sigma_{t^*}(y) = 1$ for every $t^* \in [t', t'']$. Let $F(t')$ be the indicator random variable for the event that s^* appears in round t' for the first time. Hence, we get

$$\begin{aligned}
1/S - \delta &\leq \Pr[A \wedge \bar{B}] \leq \sum_{0 < t' < t - t/(3S)} \Pr[F(t') \wedge (\exists t'' \in [t' + t/3S, t] \text{ s.t } s_{t''} = s^*) \wedge \bar{B}] \\
&\leq \sum_{0 < t' < t - t/(3S)} \Pr[F(t') \wedge (\exists t'' \in [t' + t/3S, t] \text{ s.t } A(t', t'') = 1)] \\
&= \sum_{0 < t' < t - t/(3S)} \Pr[F(t')] \cdot \Pr[(\exists t'' \in [t' + t/3S, t] \text{ s.t } A(t', t'') = 1) \mid F(t')]
\end{aligned}$$

$$\begin{aligned} &= \sum_{0<t'<t-t/(3S)} \Pr[F(t')] \cdot \Pr[(\exists t'' \in [t'+t/3S, t] \text{ s.t } A(t',t'')=1) \mid s_{t'} = s^*] \\ &= \sum_{0<t'<t-t/(3S)} \Pr[F(t')] \cdot p^* \le p^*, \end{aligned}$$

where the second inequality is by union bound over all possibilities for event A. The third equation is due to the memoryless property, the probability that the event occurred conditioning on $F(t')$, is equivalent to conditioning on $s_{t'} = s^*$. The last equality follows by summing over a set of disjoint events $F(t')$. □

We are now ready to complete the proof of Theorem 3.

Proof of Theorem 3. We bound the probability that y fires in each of the first $c \cdot t$ rounds. Let C be the indicator random variable for the event that there exists a sequence of rounds $0 = \tau_0 < \tau_1 < \tau_2 < \cdots < \tau_{3c \cdot S}$ such that for every $i \ge 1$, it holds that:

- $\sigma_{t^*}(y) = 1$ for all $t^* \in [\tau_{i-1}, \tau_i]$.
- $s_{\tau_i} = s^*$.
- $\tau_i - \tau_{i+1} \in [t/3S, t]$.

Note that because $\tau_{i+1} - \tau_i \ge t/3S$, it holds that $\tau_{3c \cdot S} \ge c \cdot t$. Hence, the probability that y fires in each of the first $c \cdot t$ rounds is at least $\Pr[C]$. Next, we calculate the probability of the event C. Recall that given that $s_{\tau_i} = s^*$, the probability there exists a round $\tau_{i+1} \in [\tau_i + t/3S, \tau_i + t]$ for which $A(\tau_i, \tau_{i+1})$ is equal to $p(\tau_i) = p^*$. Moreover, by Claim 5 and the success guarantee, the probability there exists $0 < \tau_1 < t$ such that $A(0, \tau_1)$ is at least $1/S - \delta$. By Claim 6 and the memoryless property, we have:

$$\begin{aligned} \Pr[C] &= \Pr[\exists \tau_1 < t \; s.t \; A(0,\tau_1)] \cdot \prod_{i=1}^{3c \cdot S-1} \Pr[\exists \tau_{i+1} \in [\tau_i + t/3S, \tau_i + t] \; s.t \; A(\tau_i, \tau_{i+1}) \mid s_{\tau_i = s^*}] \\ &\ge (1/S - \delta) \prod_{i=1}^{3c \cdot S-1} p^* \ge (1/S-\delta)^{3cS} \; . \end{aligned}$$

Taking $N \le (\log\log 1/\delta - \log\log\log 1/\delta) - \log 6c = \log(\frac{\log 1/\delta}{\log\log 1/\delta}) - \log 6c$, the number of different states is bounded by $S < \frac{\log 1/\delta}{6c \cdot \log\log 1/\delta}$. Thus, the network fails with probability of at least

$$(1/S - \delta)^{3c \cdot S} > \left(\frac{6 \cdot c \cdot \log\log 1/\delta}{\log 1/\delta} \cdot \frac{1}{2} \right)^{\frac{\log 1/\delta}{\log\log 1/\delta}} > \delta \; ,$$

in contradiction to the success guarantee of at least $1 - \delta$.

2.4 Applications to Synchronizers

The asynchronous setting. In this setting, the neural network $\mathcal{N} = \langle X, Z, Y, w, b\rangle$ also specifies a response latency function $\ell : V \times V \to \mathbb{N}_{>0}$, where $V = X \cup Z \cup Y$. For ease of notation, we normalize all latency values such that $\min_{e\in A} \ell(e) = 1$ and denote the maximum response latency by $L = \max_{e\in A} \ell(e)$. Supported by biological evidence [91], we assume that self-loop edges (a.k.a. autapses) have the minimal latency in the network, that $\ell((u, u)) = 1$ for self-edges (u, u). This assumption is crucial in our design[1]. Indeed the exceptional short latency of self-loop edges has been shown to play a critical role in biological network synchronization [125, 62]. The dynamic proceeds in synchronous rounds and phases. The length of a round corresponds to the minimum edge latency, this is why we normalize the latency values so that $\min_{e\in A} \ell(e) = 1$. If neuron u fires in round τ, its endpoint v receives u's signal in round $\tau + \ell(e)$. Formally, a neuron u fires in round τ with probability $p(u, \tau)$:

$$\mathrm{pot}(u, \tau) = \sum_{v\in X\cup Z\cup Y} w_{v,u} \cdot \sigma_{\tau-\ell(u,v)}(v) - b(u) \text{ and } p(u, \tau) = \frac{1}{1 + e^{-\frac{\mathrm{pot}(u,\tau)}{\lambda}}} \tag{2.1}$$

Synchronizer. A synchronizer ν is an algorithm that gets as input a network $\mathcal{N}_{\mathsf{sync}}$ and outputs a network $\mathcal{N}_{\mathsf{async}} = \nu(\mathcal{N}_{\mathsf{sync}})$ such that $V(\mathcal{N}_{\mathsf{sync}}) \subseteq V(\mathcal{N}_{\mathsf{async}})$ where $V(\mathcal{N})$ denotes the neurons of a network $\mathcal{N}$. The network $\mathcal{N}_{\mathsf{async}}$ works in the asynchronous setting and should have *similar execution* to $\mathcal{N}_{\mathsf{sync}}$ in the sense that for every neuron $v \in V(\mathcal{N}_{\mathsf{sync}})$, the firing pattern of v in the asynchronous network should be similar to the one in the synchronous network. The output network $\mathcal{N}_{\mathsf{async}}$ simulates each round of the network $\mathcal{N}_{\mathsf{sync}}$ as a *phase*.

Definition 3 (Pulse Generator and Phases)**.** *A* pulse generator *is a module that fires to declare the end of each phase. Denote by $t(v, p)$ the (global) round in which neuron v receives the p-th spike from the pulse generator. We say that v is in phase p during all rounds $\tau \in [t(v, p - 1), t(v, p)]$.*

Definition 4 (Similar Execution (Deterministic Networks))**.** *The* synchronous *execution* Π_{sync} *of a deterministic network $\mathcal{N}_{\mathsf{sync}}$ is specified by a list of states $\Pi_{\mathsf{sync}} = \{\sigma_1, \ldots, \}$ where each σ_i is a binary vector describing the firing status of the neurons in round i. The* asynchronous *execution of network $\mathcal{N}_{\mathsf{async}}$ denoted by Π_{async} is defined analogously only when applying the asynchronous dynamics (of Eq. (2.1)). The execution Π_{async} is divided into phases of fixed length. The networks $\mathcal{N}_{\mathsf{sync}}$ and $\mathcal{N}_{\mathsf{async}}$ have a* similar execution *if $V(\mathcal{N}_{\mathsf{sync}}) \subseteq V(\mathcal{N}_{\mathsf{async}})$, and in addition, a neuron $v \in V(\mathcal{N}_{\mathsf{sync}})$ fires in round p in the execution Π_{sync} iff v fires during phase p in Π_{async}.*

For simplicity of explanation, we assume that the network $\mathcal{N}_{\mathsf{sync}}$ is deterministic. However, our scheme can easily capture randomized networks as well (i.e., by fixing the random bits in the synchronized simulation and feeding it to the async. one).

[1]In [89], we actually show that this assumption is necessary for the existence of synchronizers even when $L = 2$.

Extension for randomized networks. For networks $\mathcal{N}_{\mathsf{sync}}$ that contain also probabilistic threshold gates, the notion of similar execution is defined as follows. Consider a fixed execution Π_{sync} of the network $\mathcal{N}_{\mathsf{sync}}$. In each round of simulating $\mathcal{N}_{\mathsf{sync}}$, the spiking neurons flip a coin with probability that depends on their potential. Once we fix those random coins used by the neurons in the execution Π_{sync}, the process becomes deterministic. Formally, for every round p and neuron v, let $R(v,p)$ be the set of random coins used by the neuron v in round p in the execution Π_{sync}. The firing decision of v in round p is fully determined given those bits. The asynchronous network $\mathcal{N}_{\mathsf{async}}$ contains a set of neurons V' that are analogous to the neurons in $\mathcal{N}_{\mathsf{sync}}$ and an additional set of deterministic threshold gates. When simulating this network, the neurons in V' will use the *same* random coins as those used by their corresponding neurons in Π_{sync}: in each phase p in the execution Π_{async}, the neuron v will be given the bits $R(v,p)$ and will base its firing decision using a deterministic function of its current potential, bias value and $R(v,p)$. This allows us to restrict attention to deterministic networks[1].

The challenge. Consider a (synchronous) network that consists of one output neuron z with two incoming inputs: an excitatory neuron x, and an inhibitory neuron y. The weights are set such that z computes $X \wedge \overline{Y}$. Thus, z fires in round τ if x fired in round $\tau - 1$ and y did not fire. Implementing an $X \wedge \overline{Y}$ gate in the asynchronous setting is quite tricky. In the case where both x and y fire in round τ, in the synchronous network, z should not fire in round $\tau + 1$. However, in the asynchronous setting, if $\ell(x,z) < \ell(y,z)$, then z will mistakenly fire in round $\tau + \ell(x,z)$. This illustrates the need of enforcing a *delay* in the asynchronous simulation: the neurons should attempt firing only after receiving *all* their inputs from the previous phase. We handle this by introducing a pulse-generator module, that announces when it is safe to attempt firing.

To illustrate another source of challenge, consider the asynchronous implementation of an AND-gate $X \wedge Y$. If both x and y fire in round τ, then z fires in round $\tau + 1$ in the synchronous setting. However, if the latencies of the edges $\ell(x,z)$ and $\ell(y,z)$ are distinct, z receives the spike from x and y in *different* rounds, preventing the firing of z. Recall, that z has no memory, and therefore its firing decision is based only on the potential level in the previous round. To overcome this hurdle, in the transformed network, each neuron in the original synchronous network is augmented with 3 copy-neurons, some of which have self-loops. Since self-loops have latency 1, once a neuron with a self-loop fires, it fires in the next round as well. This will make sure that the firing states of x and y are kept on being presented to z for sufficiently many rounds, which guarantees the existence of a round where both spikes arrive.

While solving one problem, introducing self-loops into the system brings along other troubles. Clearly, we would not want the neurons to fire forever, and at some point, those neurons should get inhibited to allow the beginning of a new phase. This calls for a delicate *reset* mech-

[1]The neurons in those networks are not necessarily threshold gates, but rather base their firing decision using *some* deterministic function

anism that cleans up the old firing states at the end of each phase, only after their values have already been used. Our final solution consists of global synchronization modules (e.g., pulse-generator, reset modules) that are inter-connected to a modified version of the synchronous network. Before explaining those constructions, we start by providing a modified neural timer $\mathsf{DetTimer}_{\mathsf{async}}$ adapted to the asynchronous setting. This timer will be the basic building block in our global synchronization infrastructures.

Asynchronous analog of $\mathsf{DetTimer}$**.** A basic building block in our construction is a variant of $\mathsf{DetTimer}$ to the asynchronous setting. Observe that the $\mathsf{DetTimer}$ implementation of Section 2.2 might fail miserably in the asynchronous setting, e.g., when the edges $(a_{i-1,2}, a_{i,2})$ have latency 2 for every $i \geq 2$, and the remaining edges have latency 1, the timer will stop counting after $\Theta(\log t)$ rounds, rather than after t rounds.

Lemma 3 (Neural Timer in the Asynchronous Setting)**.** *For a given time parameter t, there exists a deterministic network $\mathsf{DetTimer}_{\mathsf{async}}$ with $O(L \cdot \log t)$ neurons, satisfying that in the asynchronous setting with maximum latency L, the output neuron fires at least $\Theta(t)$ rounds, and at most $\Theta(L \cdot t)$ rounds after each firing of the input neuron.*

The construction starts with $t' = t/2L$ layers of the $\mathsf{DetTimer}$ network. These layers are modified as follows (see Fig. 2.6 for comparison with the standard construction).

- Neurons $a_{1,1}$ and $a_{1,2}$ are connected by a chain of length $4L$. all neurons in the chain as well as $a_{1,2}$ have an incoming edge from the previous neuron in the chain with weight 1 and threshold 1.

- For every $i \geq 2$, the inhibitor neuron d_i has an incoming edge only from $a_{i,2}$ with weight $w(a_{i,2}, d_i) = 1$ and threshold $b(d_i) = 1$.

- For every $i \geq 1$, the neurons $a_{i-1,2}$ and $a_{i,1}$ are connected by a chain of length L, instead of a direct edge, where the weight of the edge from the end of the chain to $a_{i,1}$, is 1.

- The neuron $a_{\log t',2}$ is an excitatory (rather than an inhibitory) neuron, and the output neuron y has one incoming edge from $a_{\log t',2}$ with weight $w(a_{\log t',2}, y) = 1$ and threshold $b(y) = 1$.

- A newly introduced inhibitor neuron r that has an incoming edge from $a_{\log t',2}$ with weight $w(a_{\log t',2}, r) = 1$, threshold $b(r) = 1$, and negative outgoing edges to all neurons in the timer with weight -2 for clean-up purpose.

The correctness is based on the following auxiliary claim.

Claim 7. *Fix a layer $i \geq 2$. Assume that (1) $a_{i-1,2}$ fired for the first time in round f_{i-1}, and that (2) it fires every τ_{i-1} rounds. It then holds that (1a) $a_{i,2}$ fires for the first time in round f_i for $f_i \in [f_{i-1} + \tau_{i-1} + 1, f_{i-1} + \tau_{i-1} + L^2 + L]$, and that (1b) $a_{i,2}$ fires from that point on for every $\tau_i \in [2 \cdot \tau_{i-1}, 2 \cdot \tau_{i-1} + (L^2 + L)]$ rounds.*

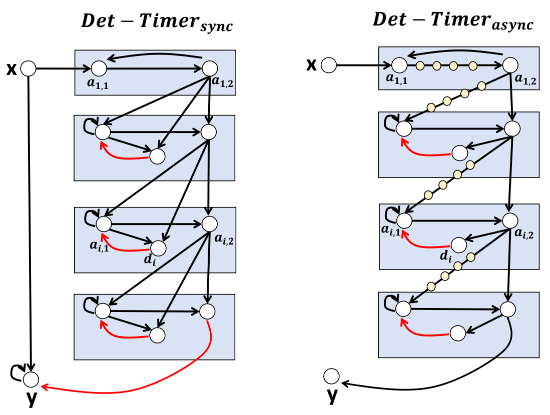

Figure 2.6: $\mathsf{DetTimer}_{\mathsf{async}}$ versus $\mathsf{DetTimer}_{\mathsf{sync}}$. Left: The deterministic timer $\mathsf{DetTimer}_{\mathsf{sync}}$ network. Right: The modified $\mathsf{DetTimer}_{\mathsf{async}}$ network which works in the asynchronous setting. We add a chain of L neurons in the first layer and between neurons $a_{i-1,2}$ and $a_{i,1}$, where L is the upper bound on the response latency of a single edge in the asynchronous setting.

Proof. Assume that neuron $a_{i-1,2}$ fires every τ_{i-1} rounds starting round f_{i-1}. It then holds that $a_{i,2}$ gets the spike from $a_{i-1,2}$ strictly *before* the spike of $a_{i,1}$. Specifically, it gets the spike from $a_{i-1,2}$ by round $\tau \leq f_{i-1}+L$, and it receives the spike from $a_{i,1}$ in some round $\tau' \geq f_{i-1}+L+1$. Note that it is crucial that the spike from $a_{i-1,2}$ arrives earlier to $a_{i,2}$, as otherwise $a_{i,2}$ will fire in round τ. As a result, the first time $a_{i,2}$ fires is after round $f_{i-1}+\tau_{i-1}+1$ and therefore $f_i \geq f_{i-1}+\tau_{i-1}+1$. Due to the self loop on $a_{i,1}$, neuron $a_{i,2}$ gets a spike from $a_{i,1}$ in every round $\tau'' \geq \tau'$. Because the latencies are fixed, $a_{i,2}$ gets a signal from $a_{i-1,2}$ every τ_{i-1} rounds, and therefore $a_{i,2}$ fires by round $\tau' + \tau_{i-1}$. Since each edge has latency of at most L, it holds that $\tau' \leq f_{i-1}+L^2+L$, hence $f_i \leq f_{i-1}+L^2+L+\tau_{i-1}$ and (1a) follows.

We now show (1b). We first observe that $a_{i,1}$ stops firing at least L rounds *before* the next firing of $a_{i-1,2}$. This holds since once $a_{i,2}$ fires in round f_i, after at most L rounds the inhibitor d_i fires, and after at most $2L$ rounds neuron $a_{i,1}$ is inhibited. Since $\tau_j \geq 4L$ (due to the chain in the first layer), it indeed holds that in the next round when $a_{i-1,2}$ fires, no neuron in layer i fires. Since the latency of each edge is fixed and $a_{i-1,2}$ fires every τ_{i-1} rounds by our assumption, we conclude that $a_{i,2}$ fires every τ_i rounds where $\tau_i \in [2\tau_{i-1}, 2\tau_{i-1}+L^2+L]$. □

Claim 8. *Assume that x fired in round τ_0. Then for every $i \geq 1$ it holds that: (1) the neuron $a_{i,2}$ fires for the first time during the interval $[\tau_0+2^i\cdot 2L, \tau_0+2^i\cdot 8L^2]$ and (2) it fires every τ_i rounds for $\tau_i \in [2^i\cdot 2L, 2^i\cdot(4L^2)]$.*

Proof. Once the input neuron x fired in round τ_0, the neuron $a_{1,2}$ fires for the first time in round $f_1 \in [\tau_0+4L, \tau_0+4L^2+L]$ and continue to fire every τ_1 rounds for $\tau_1 \in [4L, 4L^2+L]$. This is due to the chain between $a_{1,1}$ and $a_{1,2}$ and the fact that the latency $\ell(e)$ is fixed for every e. Using Claim 7 in an inductive manner, we conclude that for every $i \geq 1$: (1) $a_{i,2}$ fires every

$\tau_i \in [2^i \cdot 2L, 2^i \cdot 4L^2]$ rounds, (2) $a_{i,2}$ fires for the first time in round $f_i \in [\tau_0 + 2^i \cdot 2L, \tau_0 + 2^i \cdot 8L^2]$.
□

Proof of Lemma 3. Since the edge between neuron $a_{\log t',2}$ and the output neuron has latency at most L, we conclude that if the input neuron x fires in round τ_0, the output neuron y fires in round $\tau \in [\tau_0 + 2Lt', \tau_0 + 9L^2 f']$. Because $t' = t/2L$, given that the input x fired in round τ, the output neuron fires between round $\tau + t$ and round $\tau + 5Lt$. Lemma 3 follows. □

The synchronizers description. The construction has two parts: a global infrastructure, that can be used to synchronize many networks[1], and an adaptation of the given network $\mathcal{N}_{\mathsf{sync}}$ into a network $\mathcal{N}_{\mathsf{async}}$. The global infrastructure consists of the following modules:

- A *pulse generator* PG implemented by $\mathsf{DetTimer}_{\mathsf{async}}$ with time parameter $\Theta(L^3)$.
- A *reset* module R_1 implemented by a directed chain of $\Theta(L)$ neurons [2] with input from the output neuron of the PG module.
- A *delay* module D implemented by $\mathsf{DetTimer}_{\mathsf{async}}$ with time parameter $\Theta(L^2)$ and input from the output of of the PG module.
- Another *reset* module R_2 implemented by a chain of $\Theta(L)$ neurons with input from D.

The heart of the construction is the pulse-generator that fires once within a fixed number of $\ell \in [\Theta(L^3), \Theta(L^4)]$ rounds, and invokes a cascade of activities at the end of each phase. When its output neuron g fires, it activates the reset and the delay modules, R_1 and D. The second reset module R_2 will be activated by the delay module D. Both reset modules R_1 and R_2 are implemented by chains of length L, with the last neuron on these chains being an *inhibitor* neuron. The role of the reset modules is to *erase* the firing states of some neurons (in $\mathcal{N}_{\mathsf{async}}$) from the previous phase, hence their output neuron is an inhibitor. The timing of this clean-up is very delicate, and therefore the reset modules are separated by a delay module that prevents a premature operation. The total number of neurons in these global modules is $O(L \cdot \log L)$. We next consider the specific modifications to the synchronous network $\mathcal{N}_{\mathsf{sync}}$ (see Fig. 2.7).

Modifications to the network $\mathcal{N}_{\mathsf{sync}}$. The input layer and output layer in N_{async} are *exactly* as in $\mathcal{N}_{\mathsf{sync}}$. We will now focus on the set of *auxiliary* neurons V in $\mathcal{N}_{\mathsf{sync}}$. In the network $\mathcal{N}_{\mathsf{async}}$, each $v \in V$ is augmented by three additional neurons $v_{\mathsf{in}}, v_{\mathsf{delay}}$ and v_{out}. The incoming (resp., outgoing) neighbors to v_{in} (resp., v_{out}) are the out-copies (resp., in-copies) of all incoming (resp., outgoing) neighboring neurons of v. The neurons $v_{\mathsf{in}}, v, v_{\mathsf{delay}}$ and v_{out} are connected by a directed chain (in this order). Both v_{delay} and v_{out} have self-loops.

[1]It is indeed believed that the neural brain has centers of synchronization.

[2]Each neuron in the chain has an incoming edge from its preceding neuron with weight 1 and threshold 1.

In the case where the original network $\mathcal{N}_{\text{sync}}$ contains spiking neurons, the neuron v_{in} will be given the exact same firing function as v in Π_{sync}. That is, in phase p, v_{in} will be given the random coins[1] used by v in round p in Π_{sync}. The other neurons v, v_{delay} and v_{out} are deterministic threshold gates. The role of the *out-copy* v_{out} is to *keep on presenting* the firing status of v from the previous phase $p-1$ throughout the rounds of phase p. This is achieved through their self-loops. The role of the *in-copy* v_{in} is to simulate the firing behavior of v in phase p. We will make sure that v_{in} fires in phase p only if v fires in round p in Π_{sync}. For this reason, we set the incoming edge weights of v_{in} as well as its bias to be exactly the same as that of v in $\mathcal{N}_{\text{sync}}$. The neuron v is an AND gate of its in-copy v_{in} and the PG output g. Thus, we will make sure that v fires at the *end* of phase p only if v_{in} fires in this phase as well. The role of the delay copy v_{delay} is to delay the update of v_{out} to the up-to-date firing state of v (in phase p). Since both neurons v_{delay} and v_{out} have self-loops, at the end of each phase, we need to carefully reset their values (through inhibition). This is the role of the reset modules R_1 and R_2. Specifically, the reset module R_1 operated by the pulse-generator inhibits v_{out}. The second reset module R_2 inhibits the delay neuron v_{delay} only after we can be certain that its value has already been “copied” to v_{out}. Finally, we describe the connections of the neuron v_{out}. The neuron v_{out} has an incoming edge from the reset module R_1 with a super-large weight. This makes sure that when the reset module is activated, v_{out} will be inhibited shortly after. In addition, it has a self-loop also of large weight (yet smaller than the inhibition edge) that makes sure that if v_{out} fires in a given round, and the reset module R_1 is not active, v_{out} also fires in the next round. Lastly, if v_{out} did not fire in the previous round, then it fires when receiving the spikes from *both* the delay module and from the delay-copy v_{delay}. This will make sure that the firing state of v_{delay} will be copied to v_{out} only after the output of the delay module D fires.

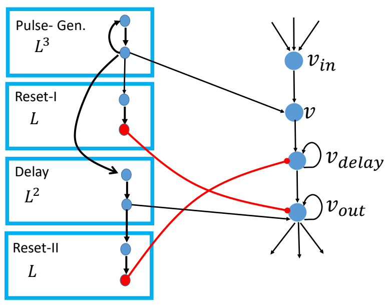

Figure 2.7: Illustration of the synchronizer modules. Left: global modules implemented by neural timers. Right: a neuron $v \in N_{\text{sync}}$ augmented by three additional neurons that interact with the global modules.

Analysis. Throughout, we fix a synchronous execution Π_{sync} and an asynchronous execution Π_{async}. For every round p, recall that $V^{+}_{\text{sync}}(p)$ is the set of neurons that fire in round p in Π_{sync} (i.e., the neurons with positive entries in σ_p). In our simulation, we will make sure that each v in $\mathcal{N}_{\text{async}}$ has the same firing pattern as its copy in $\mathcal{N}_{\text{sync}}$.

[1] I.e., the random coins that are used to simulate the firing decision of v.

Observation 4. *Consider a neuron v with incoming neighbors $u_1, \ldots, u_k$. If there is a round τ such that $u_1, \ldots, u_k$ fire in each round $\tau' \geq \tau$, v fires in every round $\tau'' \geq \tau + \max_{u_i} \ell(v, u_i)$.*

Lemma 4. *The networks $\mathcal{N}_{\mathsf{sync}}$ and $\nu(\mathcal{N}_{\mathsf{sync}}) = \mathcal{N}_{\mathsf{async}}$ have similar executions.*

In order to prove Lemma 4, we will show by induction on p that $V^+_{\mathsf{sync}}(p) = V^+_{\mathsf{async}}(p)$.
Base case. For $p = 1$, let $V^+_{\mathsf{sync}}(0)$ be the neurons that fired at the beginning of the simulation in round 0. We now show that every neuron $v \in V$ fires at the end of phase 1 iff $v \in V^+_{\mathsf{sync}}(1)$. Without loss of generality, assume that g fired at the end phase 0 and begins the simulation in round 0. We begin with the following claim.

Claim 9. *For every $u \in V$, for its in-copy u_{in} there is a round $\tau_u \leq c_2 L^3 + L$ in which all its incoming neighbors in $V^+_{\mathsf{sync}}(0)$ fire (and the remaining neighbors do not fire), for a constant c_2.*

Proof. We first show that for $v \in V^+_{\mathsf{sync}}(0)$, the out-copy v_{out} fires when it receives a signal from the delay module D. Because each edge has latency at most L, by round L, neuron v has fired. Since the delay neuron v_{delay} has a self-loop (with latency one), it starts firing in every round starting round $\tau_d \in [2, 2L]$ (until it is inhibited by the reset module R_2). Recall that the out-copy v_{out} is connected to the delay module D, and fires only when receiving a spike from both the output neuron of D *and* the delay-neuron v_{delay}. We claim that v_{out} receives a signal from D and starts firing *after* it gets a reset from R_1. The reset module R_1 receives the signal from g by round L and starts counting L rounds. Thus, the output neuron of R_1 fires in some round $\tau'_{r_1} \in [L + 1, L^2 + L]$. This insures that by round $L^2 + 2L$ the neuron v_{out} is inhibited by the output of R_1. The delay module D is implemented by $\mathsf{DetTimer}_{\mathsf{async}}$ with time parameter $2L^2$. Therefore, the output neuron of D fires in round $\tau_D \in [2L^2, 10L^3]$, ensuring that it fires only *after* v_{out} has been reset by the module R_1. Moreover, the reset module R_2 counts L rounds after receiving a signal from D. This ensures that the inhibitory output of R_2 starts inhibiting v_{delay} only *after* v_{out} has received the signal from D in round τ_{out}. Overall, we conclude that v_{out} fires in round $\tau_{\mathsf{out}} \in [c_1 \cdot L^2, c_2 \cdot L^3]$, for some constants c_1, c_2. Due to the self-loop, v_{out} also fires in each round $\tau'' \geq \tau_{\mathsf{out}}$ in that phase. As a result for every $u \in V$, its in-copy u_{in} has a round $\tau_u \leq c_2 L^3 + L$ in which all its incoming neighbors in $V^+_{\mathsf{sync}}(0)$ fire. Note that for every neuron $v \notin V^+_{\mathsf{sync}}(0)$, non of its copy neurons $v_{\mathsf{out}}, v_{\mathsf{delay}}$ fire during the phase. □

Hence, u_{in} start firing in round τ_u only if u fires in round 1 in Π_{sync}, i.e., if $u \in V^+_{\mathsf{sync}}(1)$. We set the pulse-generator with time parameter $c_3 \cdot L^3$ for a large enough c_3 such that $c_3 \cdot L^3 > c_2 L^3 + 2L$. Since the out-copies keep on presenting the firing states of phase 0, u_{in} continues to fire in the last L rounds of the phase. Thus, when the pulse-generator spikes again, the neurons in $V^+_{\mathsf{sync}}(1)$ indeed fire as both g and v_{in} fired in the previous rounds. Thus, we conclude that $V^+_{\mathsf{sync}}(1) = V^+_{\mathsf{async}}(1)$.
Induction step. Assume that $V^+_{\mathsf{sync}}(p) = V^+_{\mathsf{async}}(p)$ and consider phase $p+1$. Let τ^* be the round on which the PG fired at the end of phase p. We first show the following.

Claim 10. *For every $v \in V$, the neuron v_{delay} starts firing by round $\tau^* + 2L$, iff $v \in V^+_{\mathsf{sync}}(p)$.*

Proof. Recall that all delay copies are inhibited by the reset module R_2 at most $L^2 + 2L$ rounds after the delay module D has fired. We choose the time parameter of the PG to be large enough such that this occurs before the next pulse of PG in round τ^*. Hence, when phase p ended in round τ^*, all delay copies v_{delay} are idle. Because each edge has latency of at most L, by round $\tau^* + L$, all the neurons in $V^+_{\mathsf{sync}}(p)$ have fired (and by the assumption other neurons did not fire during phase p). As a result, the neuron v_{delay} starts firing by round $\tau^* + 2L$, iff $v \in V^+_{\mathsf{sync}}(p)$. □

We next show there exists a round in which the in-copies of $V^+_{\mathsf{sync}}(p+1)$ begin to fire.

Claim 11. *For every $u \in V$ for its in-copy u_{in} there is round $\tau_u \in [\tau^* + c_1 \cdot L^2, \tau^* + c_2 \cdot L^3 + L]$ in which all its incoming neighbors in $V^+_{\mathsf{sync}}(p)$ fire, and the remaining neighbors do not fire.*

Proof. The output neuron of R_1 fires in some round $\tau' \in [\tau^* + L + 1, \tau^* + L^2 + L]$, and therefore all neurons v_{out} are inhibited by round $\tau^* + L^2 + 2L$. Recall that the delay module D is implemented by $\mathsf{DetTimer}_{\mathsf{async}}$ with time parameter $2L^2$. Therefore the output neuron of D fires in round $\tau_D \in [\tau^* + 2L^2 + 1, \tau^* + 10L^2]$, ensuring D fires after v_{out} was inhibited by R_1. Recall that the reset module R_2 counts L rounds after receiving a signal from D. This ensures that the inhibitory output of R_2 starts inhibiting v_{delay} after v_{out} received the signal from D. By Claim 10 we conclude that when neuron v_{out} receives the signal from the delay module D in some round $\tau_{\mathsf{out}} \in [\tau^* + c_1 \cdot L^2, \tau^* + c_2 L^3]$, it fires iff $v \in V^+_{\mathsf{sync}}(p)$. As a result, due to the self-loops of the out-copies, u_{in} has a round $\tau_u \in [\tau^* + c_1 \cdot L^2 + 1, \tau^* + c_2 \cdot L^3 + L]$ in which all its incoming neighbors in $V^+_{\mathsf{sync}}(p)$ fire. □

Therefore, u_{in} starts firing in round τ_u only if $u \in V^+_{\mathsf{sync}}(p+1)$ and it continues firing from round τ_u ahead in that phase due to the self-loops of the out-copies of its neighbors. Since the pulse generator fires to signal the end of phase $p+1$ in round $\tau^* + c_3 L^3 > \tau^* + c_2 \cdot L^3 + 2L$, every neuron $v \in V^+_{\mathsf{sync}}(p+1)$ fires in round $t(v, p+1)$ since both g and v_{in} fired previously (and other neurons are idle). In follows that $V^+_{\mathsf{sync}}(p+1) = V^+_{\mathsf{async}}(p+1)$.

2.5 Approximate Counting

In this section, we provide improved constructions for neural counters by allowing approximation and randomness. Our construction is inspired by the *approximate counting* algorithm of Morris as presented in [140, 70] for the setting of dynamic streaming. The idea is to implement a counter which holds the logarithm of the number of spikes with respect to base $\alpha = 1 + \Theta(\delta)$. The approximate neural counter problem is defined as follows.

Definition 5 ((Approximate) Neural Counter)**.** *Given a time parameter t and an error probability δ, an approximate neural counter has an input neuron x, a collection of $\log t$ output neurons* *represented by a vector $\bar{y}$, and additional auxiliary neurons. The network satisfies that in a time window of t rounds, in every given round, the output $\bar{y}$ encodes a constant approximation of the number of times x fired up to that round, with probability at least $1 - \delta$.*

Throughout, we assume that $1/\delta < t$. For smaller values of δ, it is preferable to use the deterministic network construction of DetCounter with $O(\log t)$ neurons described in Lemma 1. For the sake of simplicity, we first describe the construction under the following promises:

- (S1) The firing events of x are sufficiently spaced in time, that is, there are $\Omega(\log t)$ rounds between two consecutive firing events.
- (S2) The state of $\bar{y}$ encodes the right approximation in every round τ such that the last firing of x occurred before round $\tau - \log r_\tau$ where r_τ is the number of x's spikes (firing events) up to round τ.

High level description. The network ApproxCounter(t, δ) consists of two parts, one for handling a small number of spikes by the input x and one for handling large counts. The first part that handles the small number of spikes is deterministic. Specifically, as long as the number of spikes generated by x is smaller than $s = \Theta(1/\delta^2)$, we count them using the exact neural counter network (Section 2.2.2), using $O(\log 1/\delta)$ neurons. We call this module *Small Counter* (SC) and it is implemented by the DetCounter network with time parameter $\Theta(1/\delta^2)$.

To handle a large number of spikes, we introduce the *Approximate Counter* (AC) implemented by DetCounter with time parameter $\log_\alpha t$. The AC module approximates the logarithm of the number of rounds x fired with respect to base $\alpha = 1 + \Theta(\delta)$. This module is randomized and provides a good estimate for the spike count given that it is sufficiently large. The idea is to update the AC module (by adding +1) upon every firing event of x with probability $\frac{1}{1+\alpha^c}$ where c is the current value stored in the counterTo do so, we have a spiking neuron a^* that has incoming edges from the output of the AC module, and fires with the desired probability. The reason we use probability $\frac{1}{1+\alpha^c}$ instead of $\frac{1}{\alpha^c}$ as suggested in Morris algorithm, is due to the sigmoid probability function of spiking neurons (see Chapter 1). Once the count is large enough (more than s), we start using the AC module. This is done by introducing an indicator neuron v_I, indicating that the small-counter is full. The neuron v_I starts firing after SC is full (finished the count), and keeps on firing due to a self-loop.

The input to AC, denoted as x_{ac} computes an AND of the input x, the spiking neuron a^*, and the indicator neuron v_I. In addition, v_I initiates a reset of the small counter SC to make sure that the output $\bar{y}$ receives only information from the large-count module AC. Fig. 2.8 provides a schematic description of the construction.

Detailed description. Let r_n be the number of times x fired in the first n rounds, and let $\alpha = 1 + \Theta(\delta)$ be the base of the counting in the approximate counting module.

- **Handling small counts.** The module *Small-Counter* (SC) is implemented by the DetCounter module with time parameter s and input from x, where $s = \frac{1}{\delta(\alpha-1)}$. Since $\alpha = 1+\Theta(\delta)$, it

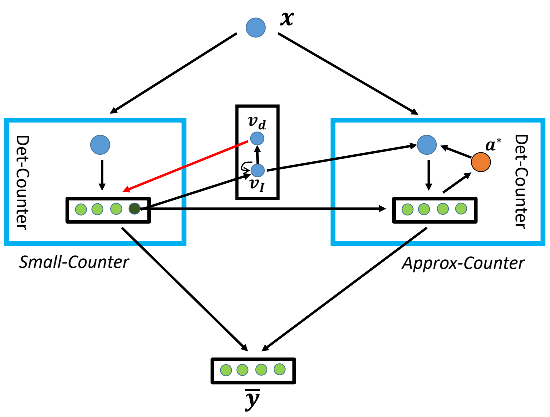

Figure 2.8: Schematic description of the network ApproxCounter. In each module, only the input and output layers are shown. The Small-Counter module SC is responsible for counting up to $\Theta(\log 1/\delta)$ spikes, and is implemented by the DetCounter module with time parameter $\Theta(1/\delta)$. For handling large counts, we use the Approx-Counter AC module implemented by the DetCounter module with time parameter $\Theta(\log t/\delta)$. The Approx-Counter module simulates Morris' algorithm and maintains an estimate for the logarithm of the spikes count. The neurons v_I and v_r switch between the two stages (small and large counts) during the execution.

holds that $s = \Theta(1/\delta^2)$. In addition, we introduce an excitatory *indicator* neuron v_I that has an incoming edge from the last layer of SC (i.e. neuron $a_{\log s,2}$) as well as a self-loop, each with weight 1 and threshold $b(v_I) = 1$. The indicator neuron v_I has an outgoing edge to an inhibitory *reset* neuron v_r with weight $w(v_I, v_r) = 1$, which is connected to all neurons in SC with negative weight -5. The reset neuron v_r also has an incoming edge from $a_{\log s,2}$ with weight 1 and threshold $b(v_r) = 1$. As a result, one round after SC reaches the value s, it is inhibited.

- **Handling large counts.** The *Approximate-Counter* (AC) is implemented by a DetCounter module with time parameter $\log_\alpha t$, and its input neuron is denoted by x_{ac}. Denote by $\ell = \log_2 \log_\alpha t$ the number of layers in the AC module, and for every $1 \leq i \leq \ell$, denote the counting neuron $a_{i,1}$ by c_i. To initialize the counter we connect the last output neuron of SC to the counting neurons $c_{i_1} \ldots c_{i_k}$ in AC which correspond to the binary representation of $\log_\alpha(1/\delta + 1)$ with weights 5. We introduce a probabilistic spiking neuron a^* that is used to increase the counter with the desired probability. In order for a^* to receive negative weights from AC, we connect each counting neuron c_i to an inhibitory copy $c_{i,2}$ with weight $w(c_i, c_{i,2}) = 1$ and threshold 1. We then connect the inhibitors $c_{1,2}, \ldots, c_{\ell,2}$ to a^* with weights $w(c_{i,2}, a^*) = -2^{i-1} \cdot \ln \alpha$, and set $b(a^*) = 0$. Hence, a^* fires in round τ with probability $\frac{1}{1+\alpha^c}$, where c is the value of AC in round $\tau - 1$. Finally, the input neuron x_{ac} has incoming edges from a^*, x and v_I each with weight 1 and threshold $b(x_{ac}) = 3$. As a result, x_{ac} fires only if v_I, x and a^* fired in the previous round.

- **The output neurons.** The counter modules SC and AC are connected to the output vector $\bar{y}$ as follows. Each y_i has incoming edges from neurons $c_1, \ldots, c_\ell$ with weight $w(c_i, y) = \log \alpha \cdot 2^{i-1}$, and threshold $b(y_i) = i + \log(\alpha - 1)$. In addition, each output neuron y_i has an incoming edge from the i^{th} output of SC with weight $b(y_i)$. Hence, y_i fires in round τ if either $\log \alpha \cdot (\sum_{j=1}^{\ell} c_j \cdot 2^{j-1}) - \log(\alpha - 1) \leq i$, or the i^{th} output of SC fired in the previous round.

Size complexity. All neurons except the spiking neuron a^* are threshold gates. Recall that $\alpha = 1 + \Theta(\delta)$. Hence the size of the counter AC is $O(\log_2 \log_\alpha t) = O(\log \log t + \log(1/\delta))$. Since the size of the counter SC is $O(\log 1/\delta)$, overall we have $O(\log \log t + \log(1/\delta))$ auxiliary neurons.

Analysis. We start with showing the correctness of the ApproxCounter network under the assumptions (S1) and (S2). At the end of the section we will show correctness for the general case as well. Let r_τ be the number of times x fired up to round τ. If $r_\tau \leq s$ the correctness of ApproxCounter follows from the correctness of the DetCounter construction (see Lemma 1). From now on, we assume $r_\tau \geq s + 1$. Let z_n be a random variable holding the value of AC after x fired n times (i.e when $r_\tau = n$). We start with bounding the expectation of α^{z_n}.

Claim 12. $\mathbb{E}[\alpha^{z_n}] \in [n(\alpha - 1)(1 - \delta) + 1, n(\alpha - 1) + 1]$.

Proof. The AC counter starts to operate after x fired $s = \frac{1}{\delta(\alpha-1)}$ spikes, and we initiate the counter with value $c = \log_\alpha(1/\delta + 1)$. Hence, for $n = s$ we get $\alpha^{z_n} = n(\alpha - 1) + 1$ and the claim holds. For $n \geq s + 1$ we get

$$\begin{aligned}
\mathbb{E}[\alpha^{z_n}] &= \sum_{j=c}^{n-1} \mathbb{E}[\alpha^{z_n} \mid z_{n-1} = j] \cdot \Pr[z_{n-1} = j] \\
&= \sum_{j=c}^{n-1} \Pr[z_{n-1} = j] \cdot (\alpha^{j+1} \cdot \frac{1}{\alpha^j + 1} + \alpha^j \cdot (1 - \frac{1}{\alpha^j + 1})) \\
&= \mathbb{E}[\alpha^{z_{n-1}}] + (\alpha - 1) \cdot \sum_{j=c}^{n-1} \Pr[z_{n-1} = j] \cdot (\frac{\alpha^j}{1 + \alpha^j}) \, . \qquad (2.2)
\end{aligned}$$

Note that for $j \geq c$, it holds that $1 > \frac{\alpha^j}{1+\alpha^j} > 1 - \delta$. Therefore

$$\sum_{j=c}^{n-1} \Pr[z_{n-1} = j] \cdot (\frac{\alpha^j}{1 + \alpha^j}) \in [1 - \delta, 1] \, .$$

By combining this with Eq. (2.2) we conclude that $\mathbb{E}[\alpha^{z_n}] \in [n(\alpha - 1)(1 - \delta), n(\alpha - 1) + 1]$. □

Claim 13. $\Pr[|\alpha^{z_n} - \mu| > 1/2 \cdot \mu] \leq \delta$, *where* $\mu = \mathbb{E}[\alpha^{z_n}]$.

Proof. We will use Chebyshev's inequality and start by computing $\mathbb{E}[\alpha^{2z_n}]$ in order to bound the variance of α^{z_n}.

$$\begin{aligned}
\mathbb{E}[\alpha^{2z_n}] &= \sum_{j=c}^{n-1} \mathbb{E}[\alpha^{2z_n} \mid z_{n-1} = j] \cdot \Pr[z_{n-1} = j] \\
&= \sum_{j=c}^{n-1} \Pr[z_{n-1} = j] \cdot \left(\alpha^{2j+2} \cdot \frac{1}{\alpha^j + 1} + \alpha^{2j} \cdot (1 - \frac{1}{\alpha^j + 1})\right) \\
&= \mathbb{E}[\alpha^{2z_{n-1}}] + \sum_{j=c}^{n-1} \Pr[z_{n-1} = j] \cdot (\frac{\alpha^{2j}(\alpha^2 - 1)}{\alpha^j + 1}) \leq \mathbb{E}[\alpha^{2z_{n-1}}] + (\alpha^2 - 1)\mathbb{E}[\alpha^{z_{n-1}}] \\
&\leq \mathbb{E}[\alpha^{2z_{n-1}}] + (\alpha^2 - 1) \cdot ((n-1)(\alpha - 1) + 1) , \qquad (2.3)
\end{aligned}$$

where Ineq. (2.3) is due to Claim 12. For $n = s$, it holds that

$$\mathbb{E}[\alpha^{2z_s}] = s^2(\alpha - 1)^2 + 2s(\alpha - 1) + 1 \leq (\alpha + 1)(\alpha - 1) \sum_{i=1}^{s} i + (\alpha - 1)(\alpha + 1)s ,$$

and combined with Eq. (2.3) we get

$$\mathbb{E}[\alpha^{2z_n}] \leq \frac{1}{2} \left(n(3\alpha^2 - \alpha^3 + \alpha - 3) + n^2(\alpha + 1)(\alpha - 1)^2\right) .$$

Therefore the variance is bounded by

$$\begin{aligned}
Var[\alpha^{z_n}] &= \mathbb{E}[\alpha^{2z_n}] - (\mathbb{E}[\alpha^{z_n}])^2 \\
&\leq \frac{1}{2} n^2(\alpha - 1)^2 \left((\alpha - 1) + \delta^2\right) + n \left((\alpha - 1)(2\alpha + 1 - a^2) + 2\alpha\delta\right) .
\end{aligned}$$

Using Chebyshev's inequality and Claim 12 we can now conclude the following:

$$\begin{aligned}
\Pr[|\alpha^{z_n} - \mu| \geq 1/2 \cdot \mu] &\leq \frac{Var[\alpha^{z_n}]}{((1/2) \cdot \mu)^2} \leq \frac{4Var[\alpha^{z_n}]}{n^2(\alpha - 1)^2(1 - \delta)^2} \\
&\leq 4\left((\alpha - 1) + \delta^2\right) + \frac{8n\,(\alpha - 1 + 2\delta)}{n^2(\alpha - 1)^2(1 - \delta)^2} , \qquad (2.4)
\end{aligned}$$

since we assume $n \geq s = 1/\delta(\alpha - 1)$ it holds that $n \leq n^2(\alpha - 1)\delta$. As a result, by Eq. (2.4) we get:

$$\Pr[|\alpha^{z_n} - \mu| \geq 1/2 \cdot \mu] \leq 4\left((\alpha - 1) + \delta^2\right) + 10\delta\,(1 + 2\delta/(\alpha - 1)) .$$

Since $\alpha = 1 + \Theta(\delta)$, we have that $Var[\alpha^{z_n}] \leq \Theta(\delta)$. We can use $\delta' = \Theta(\delta)$ in our construction and set parameter α accordingly in order to achieve

$$\Pr[|\alpha^{z_n} - \mu| \geq 1/2 \cdot \mu] \leq \delta .$$

□

Combining Claim 12 and Claim 13 we conclude that $\alpha^{z_n} \in [n(\alpha-1)/4, 2n(\alpha-1)]$ with probability at least $1-\delta$. Let $S = \log\alpha \cdot z_n - \log(\alpha-1)$. Thus, $S \in [\log(n/4), \log(2n)]$. Recall that after SC gets reset, each y_i fires only if $\log\alpha \cdot z_n - \log(\alpha-1) \leq i$. As a result, the value of the output $\bar{y}$ is given by

$$\mathsf{dec}(\bar{y}) = \sum_{i=1}^{S} 2^i = 2^{S+1} - 2 \in [n/2-2, 4n-1] ,$$

which is a constant approximation of n as desired.

Adaptation to the general case. We now explain the modifications needed to handle the general case without the two simplifying assumptions. In order to fire with the correct probability without the spacing guarantee, every time we increase AC, we wait until its value gets updated before we attempt to increase it again. In order for the output $\bar{y}$ to output the correct value also during the update of the counter AC, we introduce an intermediate layer of neurons $c''_1, \ldots, c''_\ell$ that will hold the previous state of AC during the update.

- **Removing assumption (S1):** In the DetCounter construction, we say that there are k *active layers* in round τ if the value of the counter in round τ is at most 2^k and no neuron in layer $j \geq k+1$ fired. Once we increase the counter after at most $k+1$ rounds the value is updated. During this update operation, the network waits and ignores spikes from x that might occur during this time window. To implement this waiting mechanism, we introduce a *Wait-Timer* (WT) module which uses the DetTimer* module[1]. This DetTimer* gets an input from x_{ac} and the time parameter input $\bar{q}$ with $\log\ell$ neurons where $\ell = \log_2\log_\alpha t$ is the number of layers in the module AC. The counting neurons $c_1, \ldots, c_\ell$ of AC are connected to $\bar{q}$ as follows. Each q_i has an incoming edge from $c_{2^{i-1}}$ with weight $w(c_{2^{i-1}}, q_i) = 1$ and threshold $b(q_i) = 1$. Hence, the value of $\bar{q}$ is at least $k+1$ and at most $4k$ where k is the number of active layers in AC. In order for the time parameter to stay stable throughout the update, for each q_i we add a self-loop with weight $w(q_i, q_i) = 1$. The WT module has two outputs, an inhibitor g_r which fires as long as the timer did not finish the count, and an excitatory g which fires after the count is over. We connect r_r to x_{ac} with weight $w(g_r, x_{ac}) = -5$, preventing it from firing while the counter is not updated. We connect g to an additional inhibitory neuron q_r which inhibits the time parameter neurons $q_1, \ldots, q_\ell$ one round after we finished the count. The size of WT is $O(\log\ell) = O(\log\log 1/\delta + \log\log\log t)$.

- **Removing assumption (S2):** Two copies of the counting neurons $c_1, \ldots, c_\ell$ are introduced. The first copy $c'_1, \ldots c'_\ell$ allows us to copy the state of the counter AC once its update process is complete. Each c'_i has incoming edges from c_i and the excitatory output of the WT module, each with weight $w(g, c'_i) = w(c_i, c'_i) = 1$ and threshold $b(c'_i) = 2$.

[1]Recall that DetTimer* is a variant of the neural timer in which the time parameter is given as a soft-wired input and the upper bound on this input time is hard coded in the network.

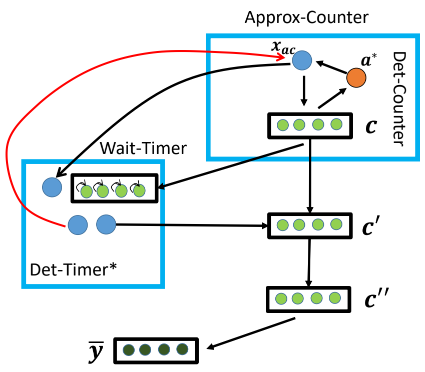

Figure 2.9: A description of the modifications in ApproxCounter (to handle the general case). The *Wait-Timer* (WT) module is implemented as a DetTimer* with input from x_{ac} and time input from the counting neurons of AC. The inhibitor output of WT inhibits the input neuron x_{ac}, preventing it from firing during the update process of the AC counter. We have two copies of the counting neurons of AC denoted as c' and c''. These copies are used for the output vector $\bar{y}$ to receive a correct input from AC at all times, even during the update process of the AC counter. Once the WT module finishes its count, in order to copy the information from c to c'', we use c' as that OR gates between c and the excitatory output of the module WT.

Thus, c'_i fires iff in the previous round both c_i and g fired (implying that neuron c_i was active when the counter finished the update). The second copy $c''_1, \ldots, c''_\ell$ holds the previous state of AC during the update of the module. Each c''_i has an incoming edge from c'_i with weight 2, a self-loop with weight 1, a negative edge from the inhibitor g_r with the weight (-1) and threshold 1. Note that the inhibition of c''_i occurs on the same round it receives the updated state from neuron c'_i. Finally, the output layer $\bar{y}$ has incoming edges from neurons $c''_1, \ldots, c''_\ell$ instead of $c_1, \ldots, c_\ell$ with the same weights.

Fig. 2.9 illustrates the modifications made to handle the general case.

Proof of Theorem 4. Assume x fired n times up to round τ. If $n \leq s$ we count the number of times x fired explicitly via the SC module. We note that in round τ the counter might be still updating the last $O(\log n)$ spikes of x. By the DetCounter construction, the value of the counter is at least $\frac{n-\log n}{2} = \Theta(n)$, and therefore we indeed output a constant approximation of n with probability 1.

Otherwise, $n \geq s$. First note that when we switch from the SC to the AC counter, we might omit at most $\Theta(\log 1/\delta)$ spikes due to the delay in the DetCounter module. Since $n \geq s = \Theta(1/\delta^2)$ this is negligible, as we want a constant approximation. Next, we bound the number of times x might have fired during the rounds in which the wait module WT was active. As we only omit attempts to increase the counter, by Claim 13 with probability at least $1-\delta$, the value of the counter has been increased for at most $\log_\alpha(2n(\alpha-1))$ times.

Each time that the counter value is increased, the waiting module WT is active for at most

$4 \log_2 \log_\alpha 2n(\alpha - 1) \leq 4 \log n$ rounds. Thus, in total we omit at most $4 \log n \cdot \log_\alpha(2n(\alpha - 1)) < 4\sqrt{n} \log^2 n$ spiking events. In addition, since the copy neurons $c''_1, \dots, c''_\ell$ might hold the previous value of the counter in round τ, we might lose another factor of two in the output layer. Altogether, in round τ the output $\bar{y}$ holds a constant approximation of n and Theorem 4 holds for the general case as well.

3 Spiking Neural Networks Through the Lens of Streaming Algorithms

3.1 Introduction

In this work, we seek to understand the role of *memory constraints* in neural data processing. We consider data-stream tasks, in which a long stream of inputs is presented over time and a neural network must evaluate some function f of this stream. Examples include identifying frequent input patterns (items) or estimating summary statistics, such as the number of distinct items presented. The network cannot store the full stream and so must maintain some form of compressed representation in its working memory, which allows the eventual computation of f. The primary objective is to compute f with as few auxiliary (non-input or output) neurons as possible. The number of auxiliary neurons can be thought of as the 'space' required by the network.

In computer science, data processing under space limitations is extensively studied in the area of streaming algorithms [142, 143]. We leverage this body of work to further our understanding of space-efficient neural networks. We start by designing neural networks for a large class of data-stream tasks, building off fundamental streaming algorithms and techniques, such as linear sketching. We also establish general connections between these models, showing that streaming-space lower bounds can be translated to neural-space lower bounds. We hope that these connections are a first step in extending work on streaming computation to better understand neural processing of massive and dynamically changing data under memory constraints.

The spiking neural network (SNN) model [128, 129]. A spiking network is represented by a directed weighted graph over n input neurons, r output neurons, and s auxiliary neurons. The edges of the graph represent synapses of different strengths connecting the neurons. The network evolves in discrete, synchronous rounds as a Markov chain where each neuron u acts as a (possibly probabilistic) threshold gate that either fires (spikes) or is silent in each round. In round t, the firing status of u depends on the firing status of its incoming neighbors in the preceding round $t-1$, and the strength of the connections from these neighbors. In *randomized* SNNs, there are possibly two sources of randomness: the spiking behavior of the neurons and the selection of random edge weights in the network. In *deterministic* SNNs, the neurons are deterministic threshold gates and the edge weights are deterministically chosen. Aside from their relevance in modeling biological computation, SNNs have received significant attention as more energy efficient alternatives to traditional artificial neural networks [111, 183].

A recent series of works in the emerging area of *algorithmic SNNs* [129, 131, 52, 119, 118, 192, 44, 112, 182, 132, 148, 89] focuses on network design tasks. In this framework, given a target function $f : \{0,1\}^n \to \{0,1\}^r$, one seeks to design a space-efficient SNN (with few auxiliary neurons) that converges rapidly to an output spiking pattern matching $f(x)$ when the input spiking pattern matches x. Space-efficient SNNs have been devised for the winner-takes-all problem [118, 182], similarity testing and compression [124, 148], clustering [84, 112], approximate counting, and time estimation [120, 86]. Interestingly, many of these works borrow ideas from related streaming algorithms. However, despite the flow of ideas from streaming to neural algorithms, the connection between these models has not been studied formally.

The streaming model [142, 143]. A data-stream is a sequence of updates $\mathcal{S} = \{u_1, \ldots, u_m\}$. A streaming algorithm $\mathcal{A}$ computes some function of $\mathcal{S}$, given restricted access to the stream. In the standard single-pass model, the algorithm can only read the updates in $\mathcal{S}$ once, in the order they are presented.

Most commonly, and throughout this work, each update u_i represents the insertion or deletion of an item x_i belonging to a universe U with $|U| = n$. Without loss of generality, we will always consider U to be the set of integers $[n] = 1, \ldots, n$, and f is a function of the frequency vector $\bar{z} \in \mathbb{Z}^n$, which tracks the total frequency of each item in the stream (the number of insertions minus the number of deletions). In the *insertion-only* setting, only insertions are allowed – i.e., each update increments some entry of $\bar{z}$. In the general *turnstile* (dynamic) setting, there are both insertions and deletions – i.e., increments and decrements to entries in $\bar{z}$. The primary complexity measure of a streaming algorithm is the *space* (measured in number of bits) required to maintain the evaluation of f on the data-stream.

Neural networks from a streaming perspective. Our primary goal is to devise space-efficient spiking neural networks that solve natural data-stream tasks, which mirror data processing tasks solved in real biological networks. In light of the large collection of space-efficient streaming algorithms that have been designed for various problems, we start by asking:

Question 1. Is it possible to translate a space-efficient streaming algorithm for a given task into a space-efficient SNN algorithm for that task? Do generic reductions from SNNs to streaming exist?

The streaming literature is also rich with space lower bounds. For many classical data-stream problems, these lower bounds are nearly tight. To obtain space lower bounds for SNNs, we ask if reductions in the reverse direction exist:

Question 2. Is it possible to translate a space-efficient SNN for a given task into a space-efficient streaming algorithm for that task?

An affirmative answer to both of these questions would imply that the streaming and SNN models are, roughly speaking, computationally equivalent. A priori, it is unclear if this is the case. On the one hand, streaming algorithms have the potential to be more space-efficient than SNNs. For example, a space-efficient algorithm may still have a lengthy description, which is not taken into account in its space complexity. In the SNN setting, where the algorithm description and memory are both encoded by the auxiliary neurons in the network and their connections, a lengthy description may lead to a large, and hence not space-efficient network.

On the other hand, SNNs have the potential to be more space-efficient than streaming algorithms. For example, a randomized SNN with a large number of input neurons but a small number of auxiliary neurons may have a large number of random bits encoded in random connections between its inputs and auxiliary neurons. These bits are not counted as part of its space complexity. In contrast, a streaming algorithm that requires persistent access to many random bits must store these bits, possibly leading to large space complexity.

3.1.1 Our Results

We take the first steps towards formally understanding the connections between streaming algorithms and spiking neural networks. The first part of this work is devoted to studying upper bounds for SNNs, addressing Question 1. We design space-efficient neural networks for a wide class of streaming problems by simulating their respective streaming algorithms. These simulations must overcome several challenges in implementing traditional algorithms in neural networks. Most notably, in an SNN, the spiking status of the auxiliary neurons encodes the working memory of the algorithm, and their connections encode the algorithm itself. A space-efficient network with few auxiliary neurons thus inherently has limited ability to express complex algorithms. In many data-stream algorithms, the target space complexity is only polylogarithmic in the input size, making this challenge significant. Additionally, unlike traditional algorithms, a neural network evolves continuously in response to its inputs. This leads to synchronization issues – for example, if an input is not presented for a sufficient number of rounds, the firing status of the network may not converge to a proper state before the next input is presented.

The second part focuses on lower bound aspects, addressing Question 2. We show that any space-efficient neural network can be translated into a space-efficient streaming algorithm, while paying a small additive term (logarithmic in the stream length/universe size). For deterministic SNNs, such a reduction is not difficult. For randomized SNNs, the reduction is more involved, as it must account for the large number of random bits that may be implicitly stored in the random edge weights of the network. Throughout, we use the $\widetilde{O}()$ notation to hides factors that are poly-logarithmic in n, m and $1/\delta$, where n is the size of the domain, m is a bound on the stream length and δ is the error parameter.

3.1.1.1 Efficient Streaming Algorithms Yield Efficient SNNs

We consider data-stream tasks in which each update is an insertion or deletion of an integer item $x \in [n]$, and f is a function of the frequency vector $\bar{z} \in \mathbb{Z}^n$ of these items. In the streaming setting, each update can be thought as an n-length vector with a single ± 1 entry, corresponding to an item insertion or deletion. In the SNN setting, each update may be encoded as the firing of one of n input neurons along with a sign neuron indicating if the update is an increment or a decrement. Or, the update may be encoded via $O(\log n)$ input neurons, indicating the item to be inserted or deleted. These different encodings correspond to different natural settings – the first corresponds to a network that collects firing statistics from a large set of inputs and the second to a network that records statistics on a large number of possible input patterns, encoded in the spiking patterns of a smaller number of input neurons.

In either case, each input is presented for some *persistence time*, a certain number of rounds in which the input is fixed to allow the network state to converge before the next input is presented.

Linear sketching. A linear sketching algorithm is a streaming algorithm in which the state of the algorithm is a linear function of the updates seen so far. In particular, the state can be represented as the multiplication of a sketching matrix $A \in \mathbb{R}^{r \times n}$ with the frequency vector $\bar{z} \in \mathbb{Z}^n$. Such algorithms have many useful properties applicable in both the turnstile setting and in distributed settings. For example, the additive nature of these algorithms allows one to split the data-stream across multiple sites, which can process the data in an independent manner. Additionally, the obliviousness of linear sketching algorithms to the ordering of the stream yields an efficient generic derandomization scheme using the Nisan's PRG for space bounded computation [93]. Linear sketching algorithms constitute the state-of-the-art algorithms for essentially all problems in the turnstile model, including heavy-hitters, coresets for clustering problems [94], and ℓ_p estimation [48]. In fact, Li, Nguyen and Woodruff [114] present a general reduction from the streaming turnstile model to linear sketching. This reduction, and its caveats have been further studied in a recent work by Kallaugher and Price [97]. Given their ubiquity in turnstile streaming algorithms, an important step in designing space-efficient SNNs for data-stream problems is an efficient implementation of linear sketching in the neural setting. We give such an implementation:

Theorem 6 (Linear Sketch). *Let $\mathcal{A}$ be an algorithm approximating a function $f(\bar{x})$ in the turnstile model using a linear sketch with an integer matrix A of size $r \times n$. Let ℓ be a bound on the* *maximum entry in $|A\bar{x}|$ for every item $\bar{x}$. There exists a network $\mathcal{N}$ with $n+1$ input neurons, $r\cdot(\lceil\log\ell\rceil+1)$ output neurons, $O(r\cdot\log\ell)$ auxiliary neurons which implements $\mathcal{A}$ in the following sense. The first n input neurons $x=(x_1,\ldots,x_n)$ represent the inserted item $[1,n]$, and the additional input neuron s indicates the sign of the update. Each input update has a persistence time of $O(\log\ell)$ rounds. The output neurons are divided into r vectors $\bar{y}_1,\ldots,\bar{y}_r$ each of length $\log\ell$, and r neurons $s_1,\ldots,s_r$. For every $i\in\{1,\ldots,r\}$, the decimal value of the binary vector $\bar{y}_i$ is equal to the absolute value of the i-th entry of $A\cdot\bar{z}$, and the sign neuron s_i indicates the sign, where $\bar{z}$ is the sum of all input items presented in the current stream.*

Theorem 6 applies to linear sketches using integer matrices, which are commonly used, see [114]. Via scaling, the construction can be extended to rational matrices as well. We note that the network of Theorem 6 does not implement the 'decoding' step which estimates $f(\bar{z})$ from $A\cdot\bar{z}$. This step depends on the problem being solved, however it is often very simple and thus implementable via a space-efficient SNN. E.g., in ℓ_p norm estimation one might just have to compute the ℓ_p norm of $A\cdot\bar{z}$ [93]. In frequency estimation, one might have to compute an average of a subset of entries in $A\cdot\bar{z}$ [39].

Beyond our generic linear sketching reduction, we give neural solutions for two challenging problems in the insertion-only model, namely, distinct elements and median estimation. These simulation results are less general and provide several tools for bypassing critical obstacles that arise in streaming to SNN reductions.

Distinct elements. In the *distinct elements problem* one must approximate the number of distinct items appearing in a data-stream with repeated items. It is well known that an exact solution by a single-pass streaming algorithm requires linear space. In fact, as we discuss later on, one can also show that the exact computation requires linear space in the SNN setting. Therefore, we restrict our attention to $(1+\epsilon)$ approximation for the number of distinct elements for any $\epsilon\in(0,1)$. This problem has been studied thoroughly in the streaming literature [40, 17, 60, 71, 98, 29, 95, 194, 9].

In this work, we provide an efficient neural implementation for the well-known LogLog streaming algorithm by [60, 71]. While the LogLog and its improved variant the hyper-Loglog algorithm provide sub-optimal space bounds, they are commonly used in practice due to their simplicity. As we will see, they are efficiently implementable in the neural setting. In addition, we provide a nearly matching space lower bound.

Theorem 7 (Neural Computation of Distinct Elements)**.** *For every $n\in\mathbb{N}$, $\epsilon,\delta\in(0,1)$, given n input neurons representing the elements in $[n]$ there exists a network $\mathcal{N}$ with $\log n$ output neurons $\widetilde{O}(1/\epsilon^2)$ auxiliary neurons, and $O(\log\log n)$ persistence time that encode the logarithm of an $(1\pm\epsilon)$ approximation of the number of distinct elements in the current stream, with probability $1-\delta$. In addition, any SNN requires $\Omega(\log n+1/\epsilon^2)$ auxiliary neurons to compute an $(1\pm\epsilon)$ approximation for the problem, with constant probability.*

The lower bound is obtained via a communication complexity reduction that mimics the corresponding streaming reduction. We note that this reduction works perfectly, i.e., without any asymptomatic loss in the space-bound (compared to the streaming bound).

Count-Min sketch. A common tool used in many of the streaming algorithms is the Count-Min sketch data structure, which maintains frequency estimates for all items in a stream. Count-Min sketch is in fact a linear sketch, and thus can be implemented via Theorem 6. However, it is not immediately clear how to implement certain important operations, like approximate frequency (count) queries via this reduction. We thus provide a direct implementation. Our implementation applies in the setting where there are $O(\log n)$ input neurons representing each insertion/deletion of an item $x \in [n]$. However, it can easily be extended to the setting in which there are n input neurons, one for each item.

Definition 6 (Count-Min Sketch [49]). *Given parameters $\epsilon, \delta \in (0,1)$, the Count-Min sketch is a probabilistic data structure that serves as a frequency table of items in a stream. It supports two operations: (i) $\mathrm{inc}(x)$ increases the frequency of x by one; (ii) $\mathrm{count}(x)$ returns an $(1+\epsilon)$ approximation of the frequency of x with probability $1-\delta$.*

For given parameters $\epsilon, \delta > 0$, the Count-Min sketch data structure contains $\ell = O(\log 1/\delta)$ hash tables $T_1, \ldots T_\ell$ each with $b = O(1/\epsilon)$ bins, and each table T_i is indexed using a different pairwise independent hash function h_i. The $\mathrm{inc}(x)$ operation applies $T_i[h_i(x)] \leftarrow T_i[h_i(x)] + 1$ for every $i \in [\ell]$. The $\mathrm{count}(x)$ operation returns $\min_{i\in[\ell]} T_i[h_i(x)]$, which is shown to provide a good approximation for the frequency of x. The Count-Min data structure is used in many streaming algorithms including heavy-hitters, range queries, quantile estimation, and more. We provide an efficient neural implementation of a Count-Min sketch data structure, and show:

Theorem 8 (Neural Implementation of Count-Min Sketch). *For every $n, m \in \mathbb{N}$ and $\epsilon, \delta \in (0,1)$ there exists a network $\mathcal{N}$ with $\log n$ input neurons, $O(1/\epsilon \cdot \mathsf{poly}(\log m, \log 1/\delta)))$ auxiliary neurons, and $\widetilde{O}(1)$ persistence time that implements a Count-Min sketch with approximation ration $(1+\epsilon)$ and success probability $1-\delta$, for an input stream of length at most m.*

Our neural implementation of the Count-Min sketch can immediately be used to give, e.g., a simple neural approximate heavy-hitters algorithm, which returns TRUE if a presented item has frequency $\geq m/k$ in a data-stream for some integer k, and FALSE if it has frequency $\leq (1-\epsilon)m/k$. Setting $\epsilon' = O(\epsilon/k)$, a $\mathrm{count}(x)$ query will return a frequency estimate $\geq m/k$ for any true heavy-hitter x and $\leq m/k$ for any x with frequency $\leq (1-\epsilon)m/k$. By keeping a counter for m using $O(\log m)$ neurons and performing a comparison operation with the output of $\mathrm{count}(x)$, we can thus solve the heavy hitters problem. Other applications of Count-Min sketch require more complex processing of the data structure's output. To illustrate how this processing can be implemented efficiently in an SNN, we detail one such application, to median approximation.

Approximate median. One of the most fundamental statistical measures of a data-stream is its quantile. The 1/2-quantile known as the median, attracts most attention in the streaming literature [142, 134, 39, 42]. Its non-linearity nature makes it considerably harder to maintain compared to its linear cousin, the mean. As in many other streaming problems, the exact computation of the median requires linear space both in the streaming and in the SNN setting (as will be discussed later on). This motivates the study of the relaxed $(1+\epsilon)$ approximation task. In the latter, the algorithm is allowed to output an item j provided that the total number of items with value at most j is in the range $[m/2 - \epsilon m, m/2 + \epsilon m]$.

Cormode and Muthukrishnan [49] presented an elegant streaming algorithm for this problem using a space of $\widetilde{O}(1/\epsilon)$ bits. The algorithm is based on the Count-Min sketch data structure, combined with a *dyadic decomposition* technique that is used in a number of other streaming algorithms. One of our key technical algorithmic contributions is in providing an efficient neural implementation of this algorithm.

Theorem 9 (Neural Computation of Approximate Median)**.** *For every $n, m \in \mathbb{N}$ and $\epsilon, \delta \in (0,1)$, there exists a neural network $\mathcal{N}_{n,m}$ solving the $(1+\epsilon)$-approximate median problem using $O(1/\epsilon \cdot \mathsf{poly}(\log m, \log n, \log 1/\delta))$ auxiliary neurons and persistence time $\widetilde{O}(1)$ with probability at least $1-\delta$.*

3.1.1.2 Streaming Lower Bounds Yield SNN Lower Bounds

Our second contribution focuses on Question 2, showing that space-efficient SNNs can be translated into space-efficient streaming algorithms, and thus that lower bounds in the streaming model imply lower bounds in the neural setting. The underlying intuition for this transformation is based on the following observation.

Observation 1. *A spiking neural network with deterministic edge weights, n input neurons, and S non-input neurons can be simulated by a streaming algorithm using S bits of space.*

In the SNN model, the spiking behavior of neurons in a given round depends only on the firing states of their incoming neighbors in the previous round. Thus, to simulate the behavior of the network as one pass over the data-stream, it is sufficient to maintain the firing states of all non-input neurons in the network, thus storing S bits of information. When the edge weights of the network are randomly sampled such a small-space simulation becomes more involved. The explicit storage of all the edge weights might be too costly since there can be $\Omega(nS + S^2)$ edges in a network with n input neurons and S non-inputs. Nevertheless, we show that a small-space simulation is still possible using a pseudorandom number generator, if we pay an additive logarithmic overhead in the length of the stream and universe size.

Theorem 10. *Any SNN $\mathcal{N}$ with n input neurons, S non-input neurons for $S = \mathsf{poly}(n)$, and $\mathsf{poly}(n)$ persistence time can be simulated over a data-stream of length m using a total space of $O(S + \log(nm))$. The success guarantee of the simulation is $1 - 1/\mathsf{poly}(n, m)$.*

Theorem 10 is a powerful tool, since it lets us apply any streaming space lower bound (of which there are many) to give an SNN lower bound, with a loss of an $O(\log(nm))$ factor.

In some cases, we can avoid this loss by more directly considering the lower bound technique. This is obtained when the streaming lower bounds are derived via a reduction to communication complexity with shared randomness that can be applied in the SNN setting with no loss. For example, using this tighter approach we show that our neural network for the distinct elements problem is nearly space-optimal (see Section 3.3).

3.1.2 Basic Tools

Our constructions are based on several neural network modules. We start by describing existing tools we will be using, and then describe additional new tools.

Neural timers and counters. For a given time parameter t, a neural timer $\mathcal{NT}_t$ is an SNN network that consists of an input neuron x, an output neuron y, and additional auxiliary neurons. The network satisfies that in every round τ, y fires in round τ iff there exists $\tau' \in [\tau - t, \tau]$ such that x fired in round τ'. It is fairly easy to design a neural timer network with $O(t)$ auxiliary neurons. [86] presented a construction of a more succinct network $\mathcal{NT}_t$ with only $O(\log t)$ neurons. In the related setting of neural counting, the network is required to encode the *number* of firing events of its input neuron within a given time window. Specifically, given a time parameter t, a *neural counter network* $\mathcal{NC}_t$ has a single input neuron x, and $\lceil \log t \rceil$ output neurons that encode the number of firing events of x within a span of t rounds.

Fact 1 ([120, 86])**.** *For every integer parameter t, there exist (i) a neural timer network $\mathcal{NT}_t$ with $O(\log t)$ neurons, and (ii) a neural counter network $\mathcal{NC}_t$ with $O(\log t)$ auxiliary neurons, such that for every round i, the output neurons encode f_i by round $i + O(\log t)$ where f_i is the number of firing events up to round i. Both networks $\mathcal{NT}_t$ and $\mathcal{NC}_t$ are deterministic.*

Maximum computation. Given a neural representation of m elements $x_1, \ldots, x_m$ in $[n]$, it is required to design a neural network that computes the maximum value $x^* = \max_i x_i$. The network has an input layer of $m \cdot \log n$ neurons that represent the elements $x_1, \ldots, x_m$, and an output layer of $\log n$ neurons that should encode the value of the maximum value x^*. Maass [131] presented a network construction with $O(m^2 + m \cdot \log n)$ auxiliary neurons. In the high level, in this network for every pair of elements x_i, x_j, there is a designated comparison neuron $c_{i,j}$ which fires if and only if $x_i \geq x_j$. The output is then computed using additional m neurons $g_1, \ldots g_m$ where g_i fires if and only if all the comparison neurons $c_{i,1}, \ldots, c_{i,m}$ fired. We have:

Fact 2 ([131])**.** *Given m vectors of neurons $\bar{x}_1, \ldots, \bar{x}_m$ each of size $\log n$, there exists a deterministic neural network with $\log n$ output neurons $\bar{y}$ and $O(m^2 + m \log n)$ auxiliary neurons, such that if the input neurons encode the values $x_1, \ldots, x_m$ in round t, the output neurons $\bar{y}$ represents the value $\max_i x_i$ by round $t + O(1)$.*

Upon very small modifications, the same network solution can be adapted to compute the minimum and the median elements. We next describe *new* tools introduced in this work which will be heavily used in our constructions.

Potential encoding. Our SNN constructions are based on a module that encodes the potential p of a given neuron x by its binary representation using $O(\log p)$ neurons. We will use this modules in the constructions of Theorem 6 and Lemma 6.

Lemma 5. *Let x be a deterministic neuron such that $\text{pot}(x, t') \leq 2^\ell$ for every $t' \in [t, t + O(\ell)]$ for some integer $\ell \in \mathbb{N}_{>0}$. There exists a deterministic network $\text{POT}_\ell(x)$ which uses ℓ identical copies of x (with the same input and bias), 2ℓ auxiliary neurons, and ℓ output neurons $y_0 \ldots y_{\ell-1}$ that encodes $\text{pot}(x, t)$ in a binary form within $O(\ell)$ rounds.*

Proof. We begin with describing the network.

- The input to the network are ℓ excitatory copies of x denoted as $x_0, \ldots, x_{\ell-1}$. Each x_i has all the incoming edges and bias as the neuron x.
- There are ℓ inhibitory neurons $r_0, \ldots r_{\ell-1}$ each with bias -1 and no incoming edges. Hence, these inhibitory neurons keep on firing on every round. Every neuron r_i is connected to x_i with weight $w(r_i, x_i) = -(2^i - 1/2)$.
- There are additional $\ell - 1$ inhibitory neurons $v_1, \ldots, v_{\ell-1}$. Each v_i has an incoming edge from x_i with weight $w(x_i, v_i) = 1$ and bias $b(v_i) = 1$. Additionally, every v_i has $i - 1$ outgoing edges to $x_0, \ldots x_{i-1}$ with weight $w(v_i, x_j) = -2^i$.
- Let $y_0, \ldots, y_{\ell-1}$ be the output neurons of the network. Each y_i has an incoming edge from x_i with weight $w(x_i, y_i) = 1$ and bias $b(y_i) = 1$. Hence, y_i fires in some round t iff x_i fired in the previous round.

See Fig. 3.1 for an illustration of the network.

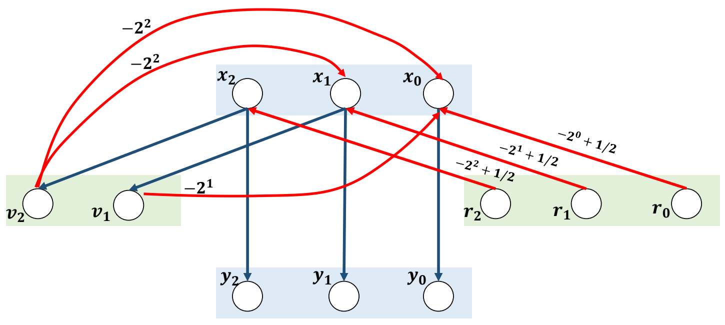

Figure 3.1: An illustration of the potential encoding module.

Correctness analysis: Let $\lfloor \text{pot}(x, t) \rfloor = \sum_{i=0}^{\ell-1} a_i \cdot 2^i$ be the potential of x in some round t_0. We assume the input is persistent for at least 2ℓ rounds. This implies that the potential of x does not change for at least 2ℓ rounds. We will show by induction on i that starting round $t_0 + 2 \cdot i$ the output neuron $y_{\ell-i-1}$ fires iff $a_{\ell-i-1} = 1$. Base case: for neuron $x_{\ell-1}$, the only inhibitor inhibiting $x_{\ell-1}$ is $r_{\ell-1}$ with weight $-(2^{\ell-1} - 1/2)$. Hence, if $a_{\ell-1} = 1$ then starting round $t_0 + 1$ the potential of $x_{\ell-1}$ is at least $\sum_{i=0}^{\ell-1} a_i \cdot 2^i - 2^{\ell-1} + 1/2 \geq 1/2 > 0$. Thus, $x_{\ell-1}$ fires starting round $t_0 + 1$ and therefore $y_{\ell-1}$ fires starting round $t_0 + 2$. On the other hand, if $a_{\ell-1} = 0$ then starting round $t_0 + 1$ the potential of $x_{\ell-1}$ is given by $\sum_{i=0}^{\ell-2} a_i \cdot 2^i - 2^{\ell-1} + 1/2 \leq (2^{\ell-1} - 1) - 2^{\ell-1} + 1/2 < 0$ and therefore starting round $t_0 + 1$ the neuron $x_{\ell-1}$ is idle and $y_{\ell-1}$ does not fire starting round $t_0 + 2$. Assume the claim is correct for neurons $y_{\ell-1}, \ldots, y_{\ell-i}$ and consider neuron $y_{\ell-i-1}$.

By the induction assumption by round $t_0+2\cdot i$ the neurons $y_{\ell-1}, \ldots y_{\ell-i}$ encode $a_{\ell-1}, \ldots a_{\ell-i}$. By the definition of the network, we conclude that for the inhibitors $v_{\ell-1}, \ldots v_{\ell-i}$, each v_j fires starting round $t_0 + 2i$ iff $a_j = 1$. Hence, the potential of x_{i-1} in round $t_0 + 2i + 1$ is equal to $\sum_{i=0}^{\ell-i-1} a_j \cdot 2^j - 2^{\ell-i-1} + 1/2$. Therefore $x_{\ell-i-1}$ fires starting round $t_0 + 2i + 1$ iff $a_{\ell-i-1} = 1$ and $y_{\ell-i-1}$ encodes a_{i-1} starting round $t_0 + 2(i + 1)$. □

Neural networks for data-stream problems. A data-stream problem is defined by a relation $P_n \subset \mathbb{Z}^n \times \mathbb{Z}$. The length of the stream is upper bounded by some integer m. Each data-item is represented by a binary vector of length n. A value $i \in [1, n]$ is represented by having the i-th input neuron fire while all other input neurons are idle. Each input is presented for some persistence time, at the end of which the output neurons of the network encode (in binary) the evaluation of a given relation over the current stream. To avoid cumbersome notation, we may assume that m and n are powers of 2.

Implementing pairwise independent hash functions. Many streaming algorithms in the insertion only model are based on the notion of pairwise independent hash functions.

Definition 7 (Pairwise Independence Hash Functions)**.** *A family of functions $\mathcal{H} : [a] \to [b]$ is* pairwise independent *if for every $x_1 \neq x_2 \in [a]$ and $y_1, y_2 \in [b]$, we have:*

$$\Pr[h(x_1) = y_1 \textit{ and } h(x_2) = y_2] = 1/b^2.$$

For ease of notation, assume that a, b are powers of 2.

Definition 8 (Pairwise Independence Hash SNN)**.** *Given two integers a, b, a* pairwise independent hash network $\mathcal{N}_{a,b}$ *is an SNN with an input layer of* $\log a$ *neurons, an output layer of* $\log b$ *neurons, and a set of s auxiliary spiking neurons. For every input value x presented at round t, let $\mathcal{N}(x)$ be the value of the output layer after $\tau_{a,b}$ rounds. Then, for every $x \neq x' \in [a]$, it holds that* $\Pr[\mathcal{N}(x) = \mathcal{N}(x')] = 1/b$.

We show a neural network implementation of a pairwise independent hash function using the construction of pairwise hash function by [36].

Lemma 6 (Neural Implementation of Pairwise Indep. Hash Function)**.** *For every integers a, b, there exists a pairwise independent hash network $\mathcal{N}_{a,b}$ with $s = O(\log b \cdot \log\log a)$ auxiliary neurons and $O(\log\log a)$ persistence time.*

Proof of Lemma 6. Our neural network implements the well-known construction of a pairwise independent hash function of [36]. In this construction, the input is a binary vector of length $c = \log a$, and the output is a binary vector of length $d = \log b$. Letting H be a binary $c \times d$ matrix sampled uniformly at random, define $h(x) = H \cdot x$, where the addition operations are defined over the field $\mathbb{F}_2$. [36] showed that for every $x \neq y$, $\Pr[h(x) = h(y)] = 1/2^d = 1/b$.

For each $i \in [\log b]$, the network connects all input neurons to one intermediate neuron r_i with random weights in $\{0, 1\}$ and bias $b(r_i) = 0$. Hence, the potential of r_i is equal to the multiplicity of $\bar{x}$ with a binary random vector. Next, the network extracts the potential of r_i using the sub-network $\mathrm{POT}(r_i)$ defined in Lemma 5. For that purpose, it introduces $O(\log \log a)$ copies of the neuron r_i. Next, in order to compute the addition in $\mathbb{F}_2$, the network computes the parity of the potential of r_i by connecting the least significant bit of the output of the sub-network $\mathrm{POT}(r_i)$ to the i-th output neuron y_i. The correctness of the constructed network follows from Lemma 5. □

3.2 Linear Sketching

A linear sketching algorithm is a streaming algorithm in which the state of the algorithm at time t is a linear function of the updates seen up to time t. We start with a formal definition.

Definition 9 (Linear Sketching Algorithm, [99]). *A linear sketching algorithm $\mathcal{L}$ gives a method for processing a vector $\bar{x} \in \mathbb{R}^n$. The algorithm is characterized by a (typically randomized) sketch matrix $A \in \mathbb{R}^{r \times n}$, and by a possibly randomized decoding function $f : \mathbb{R}^r \to O$ where O is some output domain. Alg. $\mathcal{L}$ is executed by first computing $A \cdot \bar{x}$ and then outputting $f(A \cdot \bar{x})$. Note that f only takes $A \cdot \bar{x}$ as input, f cannot depend on A in any other way, e.g. it cannot share randomness with A.*

Linear sketching algorithms provide the state-of-the-art space bounds for a large collection of problems in the turnstile model.

The challenge and our approach. Throughout we assume the sketching matrix is integral, i.e., $A \in \mathbb{Z}^{r \times n}$, which captures most of the classic implementations in the turnstile model. We start by describing a straw man approach for computing the value $A\bar{x}$ in the neural setting: Take a single-layer neural network with an input layer of length $n + 1$ and an output layer of length r. Specifically, the input layer contains n neurons $x_1, \ldots, x_n$ that represent the absolute value of the update, and an additional *sign* neuron that indicates the sign of the update. For example, an update vector $[0, 0, -1, 0]$ is represented by letting $x_3 = 1$, $s = 1$ and $x_1, x_2, x_4 = 0$. The output layer is defined by r output neurons $y_1, \ldots, y_r$. The edge weights are specified by the matrix A where $w(x_j, y_i) = A_{i,j}$. It is then easy to verify that the weighted sum of the incoming neighbors of each neuron y_j (i.e., its potential) is the value of the j-th entry in the vector $A\bar{x}$.

This naive description fails for various reasons. First, from a biological perspective, each input neuron can be either inhibitory or excitatory. This implies that the sign of the outgoing edge weights of a given neuron must be either a plus (excitatory) or a minus (inhibitory). Mathematically, this requires the sketch matrix A to be sign-consistent (i.e., the sign of all entries in a given raw are either a plus or a minus). However, in general, the given sketch matrix might not be sign-consistent. The second technicality is that the neurons $y_1, \ldots, y_n$ have a *binary* output (either firing or not) rather then an *integer* value. The third aspect to take into account is concerned with the update mechanism. Specifically, given a stream of data items, one should make sure that each data item would be processed exactly *once* by the network. This requires a more delicate update mechanism.

In the high-level, we handle the sign-consistency challenge by dividing the sketch matrix A into a non-negative matrix A^+ and a non-positive matrix A^- where $A = A^+ - A^-$. Then, given a new update $(\bar{x}, s)$, the network computes $A\bar{x}$ and $-A\bar{x}$ using $A^+\bar{x}$ and $A^-\bar{x}$. The final output $A\bar{x}$ is computed by using these values combined with the sign neuron s. To handle the second challenge, we use the module of Lemma 5 to translate the *potential* of each output neuron y_j (corresponding to the j'th bit in the sketch) into its binary representation. The output layer consists of $O(r \log n)$ output neurons that encode the value of the current r-length sketch.

Network description. Let $(\bar{x} = (x_1, \ldots, x_n), s)$, be the input neurons of the linear sketch network $\mathcal{N}$, where $\bar{x}$ represents the current update and s represents the update sign, firing if the update is negative. Let $A \in \mathbb{Z}^{r \times n}$ be the sketch matrix of the sketching algorithm to be implemented. We denote the multiplication of the input vector x' represented by $(\bar{x}, s)$ and A by $A \circ (\bar{x}, s)$. This notation is needed since $\bar{x}$ represents only the absolute value of the update. The output layer of the network consists of r vectors $\bar{y}_1, \ldots, \bar{y}_r$ and r sign-neurons $s_1, \ldots s_r$. Each vector $\bar{y}_j$ contains $O(\log n)$ neurons that are used to encode in binary the absolute value of the j-th entry in the output sketch. The sign of this entry is specified by the sign neuron s_j.

In order for the output neurons to continue presenting the correct value throughout the execution, all output neurons $\overline{y_1}, \ldots, \overline{y_r}, s_1, \ldots s_r$ have self-loops with large positive weights. For each output vector $\bar{y}_i$ with sign neuron s_i the algorithm introduces an equivalent vector of inhibitory neurons $\bar{y}'_i$ where each neuron $y'_{i,j}$ serves as an AND gate between $y_{i,j}$ and s_i. In addition, for each of the vectors $\bar{y}_i$, $\bar{y}'_i$, an inhibitor and excitatory copies are introduced, in which each neuron has the same incoming edges and biases as its corresponding neuron. These copies will assist us in case the new value after the current update will be negative. In our network description, all neurons, unless specified otherwise, are excitatory by default.

(1) Matrix multiplication. Let A^+ (resp., A^-) be the matrix containing the non-negative (resp., non-positive) entries of A, where

$$(A^+)_{i,j} = \begin{cases} A_{i,j}, & \text{if } A_{i,j} \geq 0 \\ 0, & \text{Otherwise .} \end{cases} \quad \text{and} \quad (A^-)_{i,j} = \begin{cases} -A_{i,j}, & \text{if } A_{i,j} < 0 \\ 0, & \text{Otherwise .} \end{cases}$$

Hence, both A^+ and A^- are non-negative matrices and $A\bar{x} = A^+\bar{x} - A^-\bar{x}$. The network contains two vectors of neurons a^+ and a^- of length r that are connected to the input neurons with weights $w(x_i, a^+_j) = A^+_{i,j}$, $w(x_i, a^-_j) = A^-_{i,j}$ for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, r\}$. For each neuron a^+_j and each neuron a^-_j, there are $\lceil \log \ell \rceil$ copies, in order to describe their potential by a binary vector denoted as z^+_j, z^-_j. This can be done using the potential-encoding module of Lemma 5. In addition, each vector z^+_j, z^-_j has an inhibitory vector copy z'^+_j, z'^-_j.

(2) Computing $A\bar{x}$ and $-A\bar{x}$. In order to calculate $A\bar{x}$ (and $(-A\bar{x})$), for each $i \in [r]$ the algorithm introduces two additional neurons $a_{p,i}, a_{n,i}$ such that the potential of $a_{p,i}$ equals $A\bar{x}$, and the potential of $a_{n,i}$ equals $(-A\bar{x})$. Neuron $a_{p,i}$ has incoming edges from z^+_i and the inhibitor vector z'^-_i with weights $w(z^+_i)_j, a_{p,i}) = 2^{j-1}$ and $w((z'^-_i)_j), a_{p,i}) = -2^{j-1}$. Similarly, neuron $a_{n,i}$ has incoming edges from z^-_i and the inhibitors z'^+_i with weights $w((z^-_i)_j, a_{n,i}) = 2^{j-1}$ and $w((z'^+_i)_j, a_{n,i}) = -2^{j-1}$. We then introduce $\log \ell$ copies for each $a_{p,i}$ and $a_{n,i}$, and extract their potential into binary vectors using Lemma 5. The output neurons of the sub-networks $\mathrm{POT}(a_{p,i})$ and $\mathrm{POT}(a_{n,i})$ together with the sign neuron s, are connected to four neural vectors $\bar{d}_{p,i}$, $\bar{d}'_{p,i}$, $\bar{d}_{n,i}$, $\bar{d}'_{n,i}$ as follows.

The vector $\mathrm{POT}(a_{p,i})$ is connected to $\bar{d}_{p,i}$ and $\bar{d}'_{p,i}$, where for every $j \in [\log \ell]$, the j-th neuron in $\bar{d}_{p,i}$ fires only if the j-th neuron in $\mathrm{POT}(a_{p,i})$ fire. In addition, the j-th neuron of $\bar{d}'_{p,i}$ is an AND gate of s and the j-th neuron of $\mathrm{POT}(a_{p,i})$. In a similar manner, the vector $\mathrm{POT}(a_{n,i})$ is connected to the inhibitory neurons $\bar{d}'_{n,i}$ where for every $j \in [\log \ell]$, the j-th neuron in $\bar{d}'_{n,i}$ fires only if the j-th neuron in $\mathrm{POT}(a_{p,i})$ fire. The j-th neuron of the excitatory neurons $\bar{d}_{n,i}$ is an AND gate of s and the j-th neuron of $\mathrm{POT}(a_{n,i})$.

Note that because $(A\bar{x})_i > 0$ or maybe $(-A\bar{x})_i > 0$ but not both, for every coordinate i either $\bar{d}_{p,i}$, $\bar{d}'_{p,i}$ contain firing neurons or $\bar{d}_{n,i}$, $\bar{d}'_{n,i}$ contain firing neurons but not both.

(3) Computing $A \circ (\bar{x}, s)$. For every coordinate i, there is an intermediate neuron q_i whose potential corresponds to the value of the i-th coordinate in the updated vector. The neuron q_i has positive incoming edges from the neurons in $\bar{y}_i$, $\bar{y}'_i$ with weights $w(y_{i,j}, q_i) = 2^{j-1}$ and $w(y'_{i,j}, q_i) = -2 \cdot 2^{j-1}$ respectively. Hence, in case the sign neuron s_i is idle, the output neurons contribute $\mathsf{dec}(\bar{y}_i)$ to the potential of q_i. In case the sign neuron s_i fires, the output neurons contribute $-\mathsf{dec}(\bar{y}_i)$ to the potential of q_i.

The next step is to add the value $A\bar{x}$ to the potential value of every q_i. To do that, the network connects the vectors $\bar{d}_{p,i}$, $\bar{d}'_{p,i}$, $\bar{d}_{n,i}$, $\bar{d}'_{p,i}$ to q_i in the following manner. The vectors $\bar{d}_{p,i}$, $\bar{d}'_{p,i}$ are connected to q_i with weights $w((\bar{d}_{p,i})_j, q_i) = 2^{j-1}$, and $w((\bar{d}'_{p,i})_j, q_i) = -2 \cdot 2^{j-1}$. Hence, in case where the sign neuron s is idle ,these neurons contribute $\mathsf{dec}(\bar{d}_{p,i})$ to the potential of q_i. In case where the sign neuron s fires, these neurons contribute $-\mathsf{dec}(\bar{d}_{p,i})$. Recall that the neural vectors $\bar{d}_{p,i}$, $\bar{d}'_{p,i}$ have firing neurons only if $(A \cdot \bar{x})_i > 0$, and in this case $\mathsf{dec}(\bar{d}_{p,i}) = (A \cdot \bar{x})_i$.

In the same manner, the vectors $\bar{d}_{n,i}$ and $\bar{d}'_{n,i}$ are connected to q_i with weights $w((\bar{d}_{n,i})_j, q_i) = 2 \cdot 2^{j-1}$, and $w((\bar{d}'_{n,i})_j, q_i) = -2^{j-1}$. Recall that the vectors $\bar{d}_{n,i}$, $\bar{d}'_{n,i}$ have firing neurons only if $(A \cdot \bar{x})_i < 0$ and in this case $\mathsf{dec}(\bar{d}_{p,i}) = -(A \cdot \bar{x})_i$.

Thus, by the above description the potential of q_i encodes the i-th coordinate of the updated output vector. To support the case where the potential of q_i is negative, there is an equivalent neuron q'_i whose potential is the additive inverse of the potential of q_i. This can be obtained by using the relevant excitatory and inhibitory copies of the vectors $\bar{y}_i$, $\overline{y'}_i$, $\bar{d}_{p,i}$, $\bar{d}'_{p,i}$, $\bar{d}_{n,i}$, $\bar{d}'_{p,i}$ that are connected to q'_i with the corresponding weights (as used in q_i up to a flip in the sign).

(4) Updating the output sketch (exactly once). The network updates the output vectors $\bar{y}_1, \ldots, \bar{y}_r$, $s_1, \ldots s_r$ in the following manner. Using Lemma 5 and $\log \ell$ identical copies of q_i and q'_i, it extracts their potential into vectors of neurons denoted as Q_i, Q'_i. The algorithm connects each neuron q'_i to the sign neuron s_i and the vectors Q_i, Q'_i to the output vector $\bar{y}_i$ via a delay chains of size $O(1)$, where $y_{i,j}$ has an incoming large positive weight from the j-th neurons of Q_i and Q'_i. The reason we use a delay chain is that we wish to add the update $A\bar{x}$ to the output $\bar{y}$ after the previous value is deleted, and only *once*. For that purpose, the following *reset mechanism* is added.

The algorithm connects the input neurons $\bar{x}$ to an intermediate excitatory neuron r_0 which serves as a simple OR gate. Let τ be an upper bound on the number of rounds between the first round the input $\bar{x}$ is presented and the round in which the neurons in $Q_1, \ldots Q_r$ and $Q'_1 \ldots Q'_r$ output the desired outcome as specified in Lemma 5. The neuron r_0 is connected to a chain C of size $\tau + 1$ where each neuron c_j in C has an incoming edge from c_{j-1} with weight 1 and bias 1. The neuron c_τ is then connected to the first neurons in the delay chains connected to q_i, Q_i, Q'_i for every coordinate i, where these neurons serve as AND gates of the input from c_τ and the corresponding neuron in q_i, Q_i, Q'_i. The last neuron in C, (i.e. $c_{\tau+1}$) is an inhibitory neuron with outgoing edges to all neurons in the network besides the delay chains with large negative weights (including the output neurons). Hence, once the network is reset the algorithm will update the output neurons *once*, and all neurons will be idle until the next update. Fig. 3.2 illustrates the constructed network.

Correctness. Let $\bar{x}, s$ be an update presented in round τ_0.

Observation 2. *For every coordinate $i \in [r]$, the firing neurons in z_i^+ encode the binary representation of $(A^+\bar{x})_i$, and the firing neurons in z_i^- encode $(A^-\bar{x})_i$ by round $\tau_1 = \tau_0 + O(\log \ell)$.*

Proof. Starting round $\tau_0 + 1$ due to the edge weights between $\bar{x}$ and a^+ (a^-), the potential value of $(a^+)_i$ is equal to $(A^+\bar{x})_i$ and the potential value of $(a^-)_i$ is equal to $(A^-\bar{x})_i$. Since all entrees in A^+ and A^- are non-negative, these potential values are non-negative. Hence, by Lemma 5 the neurons z_i^+ and z_i^- holds a binary representation of these potential within $O(\log \ell)$ rounds. □

We now show that the potential of q_i is equal to the updated value within $O(\log \ell)$ rounds. Let $\widehat{y}$ be the values encoded in the output neurons in round τ_0.

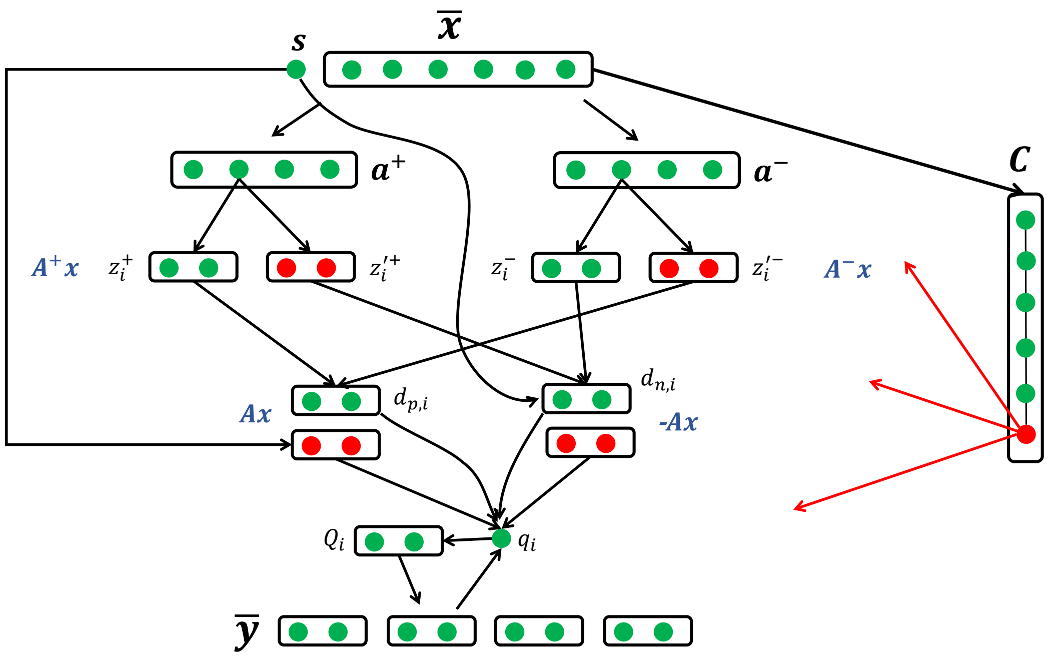

Figure 3.2: An illustration of the linear sketching network. The red circles represent inhibitor neurons, and the green circles represent excitatory neurons. In the first layer, the potential values of the vector a^+ (a^-) represent $A^+\bar{x}$ ($A^-\bar{x}$). For simplicity from that point, we focus on updating the i-th coordinate of the output neurons. In the third layer the potential value of $(A^+\bar{x})_i$ ($(A^-\bar{x})_i$) are extracted into a binary vector z_i^+ (z_i^-). These neurons are then used to compute $A\bar{x}$ and $-A\bar{x}$ which are represented in $d_{p,i}$ and $d_{n,i}$ and their inhibitory copies. The inhibitory neurons $d'_{p,i}$ and the excitatory neurons $d_{n,i}$ fire only if $s = 1$. In the next layer, these neurons and the output neurons y_i are connected to q_i such that the potential of q_i is equal to the updated value. The potential of q_i is extracted into a binary vector Q_i that is connected to the output neurons. The chain C is responsible to schedule the update so that it will occur only once, and only after the reset of y.

Claim 14. *For every coordinate $i \in [r]$, the potential of q_i is equal to the updated value $(\widehat{y} + (1-2s)A\bar{x})_i$ by round $\tau_2 = \tau_0 + O(\log \ell)$.*

Proof. By Obs. 2, the neurons z_i^+, z_i^- encodes $(A^+\bar{x})_i$ and $(A^-\bar{x})_i$ respectively by round $\tau_1 = \tau_0 + O(\log \ell)$. Since $A = A^+ - A^-$, the potential value of $a_{p,i}$ equals $(A\bar{x})_i$ and the potential value of $a_{n,i}$ is equal to $-A\bar{x})_i$ by round $\tau_1 + 1$. Hence, by Lemma 5 for some round $\tau' = \tau_1 + O(\log \ell)$, if $(A\bar{x})_i \geq 0$, the output neurons of $\mathrm{POT}(a_{p,i})$ encodes $(A\bar{x})_i$ by round τ' and if $(A\bar{x})_i < 0$ the output neurons of $\mathrm{POT}(a_{p,i})$ are idle. Similarly, if $(A\bar{x})_i \leq 0$, the output neurons of $\mathrm{POT}(a_{n,i})$ encodes $(-(A\bar{x})_i)$ by round τ' and if $(A\bar{x})_i > 0$ the output neurons of $\mathrm{POT}(a_{n,i})$ are idle.

Next, we calculate the contribution of the vectors $\bar{d}_{n,i}, \bar{d}'_{n,i}$, $\bar{d}_{p,i}, \bar{d}'_{p,i}$ to the potential of q_i. Starting at round $\tau' + 1$, if $(A\bar{x})_i \geq 0$ then $\mathsf{dec}(\bar{d}_{p,i}) = \mathsf{dec}(\mathrm{POT}(a_{p,i})) = (A\bar{x})_i$ and $\mathsf{dec}(\bar{d}_{n,i}) = \mathsf{dec}(\bar{d}'_{n,i}) = 0$. In case $s = 0$, then $\mathsf{dec}(\bar{d}'_{p,i}) = 0$ and therefore the neurons

$\bar{d}_{n,i}, \bar{d}'_{n,i}$, $\bar{d}_{p,i}, \bar{d}'_{p,i}$ contribute $(A\bar{x})_i$ to the potential of q_i. In case $(A\bar{x})_i \geq 0$ and $s = 1$ then $\mathsf{dec}(\bar{d}'_{p,i}) = (A\bar{x})_i$ and by the definition of the weights from $\bar{d}'_{p,i}$ to q_i, the neurons $\bar{d}'_{p,i}$ contribute $-2(A\bar{x})_i$ to the potential of q_i. Hence, if $(A\bar{x})_i \geq 0$ and $s = 1$ the neurons $\bar{d}_{n,i}, \bar{d}'_{n,i}$, $\bar{d}_{p,i}, \bar{d}'_{p,i}$ contribute $(A\bar{x})_i - 2(A\bar{x})_i = -(A\bar{x})_i$ to the potential.

On the other hand, in case $(A\bar{x})_i \leq 0$ then $\mathsf{dec}(\bar{d}_{p,i}) = \mathsf{dec}(\bar{d}'_{p,i}) = 0$ and $\mathsf{dec}(\bar{d}'_{n,i}) = \mathsf{dec}(\mathrm{POT}(a_{n,i})) = -(A\bar{x})_i$. If $s = 0$ then $\mathsf{dec}(\bar{d}_{n,i}) = 0$ and the neurons $\bar{d}_{n,i}, \bar{d}'_{n,i}$, $\bar{d}_{p,i}, \bar{d}'_{p,i}$ contribute $-(-(A\bar{x})_i) = (A\bar{x})_i$ to the potential of q_i. If $s = 1$ then $\mathsf{dec}(\bar{d}_{n,i}) = -(A\bar{x})_i$ and these neurons contribute $(A\bar{x})_i - 2(A\bar{x})_i = -(A\bar{x})_i$ to the potential of q_i.

Similarly, as for the output neurons $\bar{y}_i$, $\overline{y'}_i$, if $s_i = 0$, these neurons contribute $\mathsf{dec}(\widehat{y}_i)$ to the potential of q_i, and if $s_i = 1$, these neurons contribute $-\mathsf{dec}(\widehat{y}_i)$. By choosing $\tau_2 = \tau' + 2$, the claim follows. □

Claim 15. *The output neurons encode the values $\widehat{y} + (1 - 2s)A\bar{x}$ by round $\tau' = \tau_0 + O(\log \ell)$. Moreover, the output neurons continue to present the updated value until the next update presented in the input neurons.*

Proof. By Claim 14 for every coordinate i, the potential value of q_i is equal to the updated value $(\widehat{y} + (1 - 2s)A\bar{x})_i$ by round $\tau_2 = \tau_0 + O(\log \ell)$. Therefore, it also holds that the potential value of q'_i is equal to $-(\widehat{y} + (1 - 2s)A\bar{x})_i$ by round τ_2. Thus, by Lemma 5 if $(\widehat{y} + (1 - 2s)A\bar{x})_i \geq 0$ then the neurons Q_i encode $(\widehat{y} + (1 - 2s)A\bar{x})_i$ by some round $\tau_3 = \tau_2 + O(\log \ell)$ and are idle otherwise. Similarly, if $-(\widehat{y} + (1 - 2s)A\bar{x})_i \geq 0$ then Q'_i encodes $-(\widehat{y} + (1 - 2s)A\bar{x})_i$ by round τ_3. Note that either Q_i or Q'_i holds firing neurons but not both. In case Q'_i contains firing neurons, $-(\widehat{y} + (1 - 2s)A\bar{x})_i > 0$, and q'_i fires as well (in such a case the sign neuron of the i-th coordinate s_i should be updated to 1).

We set the chain C to be of size $\tau = O(\log \ell) > \tau_2 - \tau_0 + 2$. Since the first neurons in the delay chains that correspond to q'_i, Q'_i, Q_i serve as AND gates of c_τ and the corresponding neurons in q'_i, Q'_i, Q_i, they begin to fire only after the neurons q'_i, Q'_i, Q_i hold the correct values in round $\tau_0 + \tau + 1$. Additionally, due to the inhibition of $c_{\tau+1}$, starting round $\tau_0 + \tau + 2$ all neurons in the network (including y_i, s_i) are idle except the delay chains which holds the updated values. We set all the delay chains connected to the output neurons to be of size 3 and therefore in round $\tau_0 + \tau + 3$ the neurons y_i, s_i will be updated once with the correct value (after these neurons are already reset). Moreover, due to the self loops of the output neurons they will continue to present the updated value until the next update is presented. □

3.3 The Distinct Elements Problem

In the distinct elements problem, given is a stream of integers $\mathcal{S} = \{x_1, \ldots, x_m\}$ where each $x_i \in [n]$. It is then required to maintain an estimate for the number of distinct elements in the stream. We start by stating the state-of-the-art bounds for this problem in the streaming model.

Fact 3 (Streaming Space Bounds, [29],[115])**.** *For any $\epsilon, \delta \in (0,1)$, there is an $(1+\epsilon)$ approximation algorithm for distinct elements with success probability of $1-\delta$ using $O(1/\epsilon^2 \cdot \log 1/\delta + \log n)$ space. Moreover, the space bound is optimal.*

In this section, we provide a neural implementation for the well-known LogLog streaming algorithm by [60, 71]. This algorithm obtains sub-optimal space, but its simplicity makes it much more applicable in the neural setting.

Lemma 7 ([60, 71])**.** *Given a data-stream of elements in $[n]$, there exists an $(1+\epsilon)$ approximation algorithm for the distinct elements problem using $O((1/\epsilon^2 \log\log n + \log n)\log(1/\delta))$ space, with probability of $1-\delta$.*

Proof Sketch. We describe the high-level idea of the randomized LogLog algorithm under a constant success guarantee. To provide a success guarantee of $1-\delta$, the same algorithm is repeated for $O(\log(1/\delta))$ times, thus inuring an overhead of $O(\log(1/\delta))$ factor in the space complexity.

The algorithm uses a pairwise independent hash function $h : [n] \rightarrow [2^{(2\log 1/\epsilon + 3\log n)}]$ to map each input value $x_i \in [n]$ into a random string $h(x_i)$. The first $k = 2\log 1/\epsilon$ bits of $h(x_i)$ are used in order to map x_i into $2^k = 1/\epsilon^2$ buckets $b_1, \ldots, b_{2^k}$. Let $\rho(x_i)$ be the number of leading zeros in the remaining $3\log n$ bits of $h(x_i)$. In each bucket b_ℓ, the algorithm maintains the maximum value of $\rho(x_i)$ for every stream element x_i that maps into the bucket b_ℓ. Denote this maximum value by N_ℓ. The estimate on the number of distinct elements is given by $\alpha \cdot 2^k \cdot 2^{\bar{N}}$ where $\bar{N}$ is the average of the N_ℓ values over the 2^k buckets $b_1, \ldots, b_\ell, \ldots, b_{2^k}$ for some constant α. □

A neural network for the distinct element problem. We next turn to describe a neural implementation of the classical LogLog algorithm and prove Theorem 7.

Definition 10 (SNN for Distinct Elements)**.** *Given parameters n and $\epsilon, \delta \in (0,1)$, an SNN network $\mathcal{N}$ for the distinct elements problem has n input neurons $\bar{x}$, and $\log n$ output neurons $\bar{y}$. For every round t, let $f_1(t)$ be*[1] *the number of distinct elements arrived by round t. For every fixed round t, it holds that by round $\tau(t)$, the output neurons $\bar{y}$ encode the binary representation of an $(1+\epsilon)$ approximation of $f_1(t)$ with probability $1-\delta$.*

We describe the network construction based on the sequence of operations applied on the input layer. See Fig. 3.3 for an illustration.

(1) Encoding the input in a binary Form. The network contains $\log n$ neurons $x'_1, \ldots x'_{\log n}$ that represent the binary encoding of the presented element. The algorithm connects the input neurons $\bar{x}$ to the neurons $\bar{x}' = x'_1, \ldots, x'_{\log n}$ such that for every $i \in [n]$, $j \in [\log n]$ the edge

[1]We call it f_1 since the distinct elements problem computes the F_1 norm of the stream.

weight $w(x_i, x'_j) = 1$ if the j-th bit in the binary representation of i is equal to 1 and $w(x_i, x'_j) = 0$ otherwise. The bias values of these neurons are set to $b(x'_j) = 1$ for every $j \in \{1, \ldots \log n\}$.

(2) Hashing. The network contains a sub-network $\mathcal{H}$ which implements a pairwise independent hash function $h : \{0,1\}^{\log n} \to \{0,1\}^{2\log(1/\epsilon)+3\log n}$ using Lemma 6. The input to the sub-network $\mathcal{H}$ are the neurons $\bar{x}'$. Let $t = O(\log\log n)$ be an upper bound on the number of rounds required for the computation of the sub-network $\mathcal{H}$. In order to maintain the persistence of the input $\bar{x}'$ for $\Theta(t)$ rounds, the network contains a neural timer $\mathcal{NT}$ using Fact 1.

The output of $\mathcal{H}$ is denoted by $\bar{h}_b$, $\bar{h}_s$, where $\bar{h}_b$ is of length $2\log(1/\epsilon)$, and $\bar{h}_s$ is of length $3\log n$. The output neurons $\bar{h}_b$ will encode the bucket number the input is mapped to, and the vector $\bar{h}_s$ holds a binary string of size $3\log n$. The neurons $\bar{h}_b$, $\bar{h}_s$ also have inhibitory copies denoted by $\bar{h}'_b, \bar{h}'_s$.

(3) Computing the number of leading zeros. In the next step the network computes the number of leading zeros in the hashed string $\bar{h}_s$. For that purpose, we first connect the inhibitory neurons $\bar{h}'_s$ to $\bar{h}_s$ such that $\bar{h}_s$ will contain a *single* firing neuron corresponding to the leading one entry in the binary string $\bar{h}_s$. This is done by connecting each inhibitory neuron $h'_{s,i}$ to the neurons $h_{s,1}, \ldots, h_{s,i-1}$ with large negative weight. As a result, $\bar{h}_s$ contains one firing neuron such that the neuron $h_{s,i}$ fires iff the number of leading zeros in the hashed string $\bar{h}_s$ is $(3\log n - i)$. Next, the number of leading zeros in $\bar{h}_s$ is encoded into a binary form using a collection of $O(\log\log n)$ neurons denoted as $\bar{z}$, which have incoming edges from $\bar{h}_s$.

(4) Representing the $B = 1/\epsilon^2$ buckets. The buckets used in the LogLog algorithm are represented using B sets of neurons $\bar{b}_1 \ldots \bar{b}_B$, each of cardinality $\log\log n$. To maintain the value stored in each bucket, the neurons $\bar{b}_1 \ldots \bar{b}_B$ have self-loops with large positives weights. Additionally, each set of neurons $\bar{b}_i$ is connected to an inhibitor copy $\bar{b}'_i$. The invariant is that for all inputs seen so far that were mapped to bucket i, the firing state of $\bar{b}_i$ will encode the maximum number of leading zeros among all the observed strings $\bar{h}_s$.

In order to extract the index of the current bucket, the algorithm introduces B excitatory neurons $a_1, \ldots a_B$, with incoming edges from the neurons $\bar{h}_b, \bar{h}'_b$ such that a_i fires only if $\mathsf{dec}(\bar{h}_b) = i$.

(5) Comparing the number of leading zeros with the value stored in the buckets. Let $j = \mathsf{dec}(\bar{h}_b)$ be the decimal value encoded is the neurons $\bar{h}_b$ at round t. In the next step, our goal is to compare the number of leading zeros encoded in the neurons $\bar{z}$ with the value stored in the j-th bucket, encoded using the neurons $\bar{b}_j$. In order to control the precise timing of the comparison, the network introduces a *delay chain* of size $\tau = O(\log\log n)$ denoted as $C = \sigma_1, \ldots \sigma_\tau$. The first neuron in the chain σ_1 serves as an OR gate of the input neurons $\bar{x}$, and for $i = 2, \ldots, \tau$ the neuron σ_i has an incoming edge from σ_{i-1} with weight 1 and bias 1.

The network then compares the value stored in the buckets $\bar{b}_1, \ldots \bar{b}_B$ with the current value stored in $\bar{z}$ using B comparison neurons $c_1, \ldots c_B$. Each comparison neuron c_i will fire only if (i) σ_τ fired, ensuring the comparison occurs when $\bar{z}$ holds the correct value, (ii) a_i fired, indicating the input is mapped to bucket i, and (iii) $\mathsf{dec}(\bar{z}) > \mathsf{dec}(\bar{b}_i)$, indicating the current number of leading zeros is larger than the value stored in the bucket $\bar{b}_i$. This is done by setting c_i to have incoming edges from a_i and σ_τ with weight $w(a_i, c_i) = w(\sigma_\tau, c_i) = 10 \log n$, incoming edges from $\bar{z}$ with weight $\mathsf{dec}(\bar{z})$, negative incoming edges from $\bar{b}'_i$ with weight $-\mathsf{dec}(\bar{b}_i)$, and bias $b(c_i) = 20 \log n + 1$.

(6) Updating the relevant bucket. Once the comparison neuron c_j fires, the goal is to copy the new value encoded in $\bar{z}$ to the bucket $\bar{b}_j$. First, the current value stored in the bucket is deleted as follows. For every $i \in [B]$ the neuron c_i is connected to an inhibitory neuron r_i, and r_i is connected to $\bar{b}_i$ with a large negative edge weight. In order to update the value stored in the bucket *after* the deletion, the neuron c_i is connected to a chain of two neurons c_i^1 and c_i^2, such that c_i^1 has an incoming edge from c_i, and c_i^2 has an incoming edge from c_i^1. Next, for every k and i, the k-th neuron in $\bar{b}_i$ serves as an AND gate of c_i^2 and the k-th neuron z_k,

To avoid additional false updates, the neuron c_i^1 is connected to an inhibitory neuron $r_{i,2}$ that has negative outgoing edge weights to the chain C, the neurons $\bar{z}$, and the input neurons $\bar{x}$. We note that in a setting where there is a signaling neuron that fires upon the arrival of a new element, the inhibition of the input neurons can be avoided.

(7) Averaging. All neurons $\bar{b}_1, \ldots \bar{b}_B$ are connected to an intermediate neuron p such that the potential of p is set to be $\log m \cdot \alpha \cdot \sum_{i=1}^{B} \mathsf{dec}(\bar{b}_i)$, where α is a constant chosen according to the LogLog algorithm. The potential of p is encoded by the output neurons $\bar{y}$ using the $\mathrm{POT}(p)$ sub-network of Lemma 5.

(8) Amplification of the success guarantee. To amplify the success probability to $1 - \delta$, the final network consists of $k = O(\log 1/\delta)$ copies of the basic sub-network described above. The final estimation is obtained by computing the median of the k outputs of the sub-networks denoted as $\bar{y}_1, \ldots, \bar{y}_k$. This is done using a variation of the network for computing the maximum value by Maass [131] as described in Fact 2.

We are now ready to analyze the correctness of the construction, and by that complete the proof of Theorem 7.

Proof of Theorem 7. We show that the proposed network implements the LogLog algorithm of [60]. Given an input $\bar{x}$ representing an element $i \in [n]$ introduced in round τ_0, in round $\tau_0 + 1$ the neurons $\bar{x}'$ hold the binary encoding of i. Moreover, due to the neural timer $\mathcal{NT}$ connected to $\bar{x}'$, we can assume that $\bar{x}'$ keeps presenting the value i for $O(\log \log n)$ rounds.

By Lemma 6, the neurons $\bar{h}_s$ encode the output of a pairwise independent hash functions $h_1 : [n] \to [n^3]$, and $\bar{h}_b$ encodes the output of a pairwise independent hash functions $h_2 : [n] \to [1/\epsilon^2]$ by round $\tau_1 = \tau_0 + O(\log \log n)$. We next observe that the neurons $\bar{z}$ encode the number of leading zeros in $h_1(\bar{x})$ starting round $\tau_1 + 2$.

Observation 3. *Starting round $\tau_1 + 2$ it holds that* $\mathsf{dec}(\bar{z})$ *encode the number leading zeros in* $h_s(\bar{x})$.

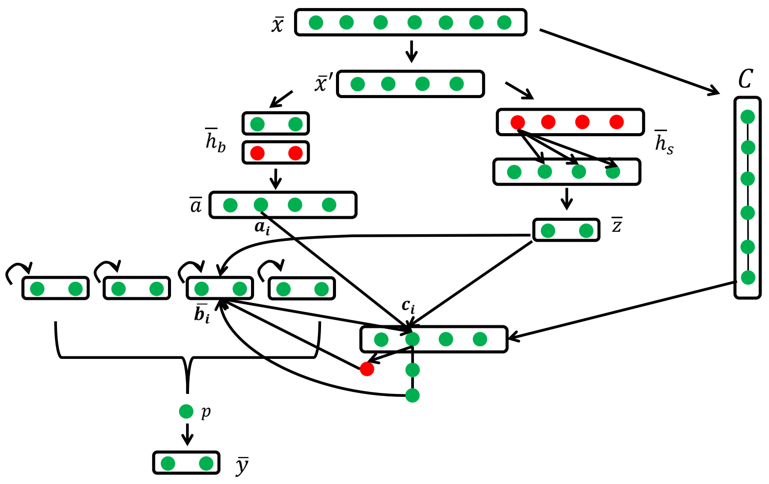

Figure 3.3: The distinct elements network. The red circles represent inhibitory neurons and the green circles represent excitatory neurons. First, the input element x is encoded into the $\log n$ neurons x' representing the value of x in binary. Then x' is hashed into two strings: (i) $\bar{h}_b$ that encodes the bucket to which x is mapped, and (ii) $\bar{h}_s$ which is the $3\log n$-bit suffix of the hash value of x. The network then encodes the number of leading zeros in the hash string $\bar{h}_s$ into a binary vector $\bar{z}$, and extracts the bucket index into a unit vector $\bar{a}$. The current maximum leading zeros in each bucket is stored in the counters on the left $(\bar{b}_1, \ldots \bar{b}_B)$. Next, the network compares between the number of leading zeros in the current string and the value stored in the relevant bucket using the comparison neurons $c_1, \ldots c_B$. The chain C is responsible for scheduling the comparison to occur only after the updated value has been computed. If needed, the corresponding bucket $\bar{b}_i$ is updated with the value encoded in $\bar{z}$ using the neurons connected to c_i.

Proof. By Lemma 6, the neurons $\bar{h}_s$ encode $h_1(\bar{x})$ by round τ_1. Due to the inhibition of $\bar{h}'_s$ in Step 3, starting round $\tau_1 + 1$ the only neuron firing in $\bar{h}_s$ is the leading one in the binary representation of $h_1(\bar{x})$. Hence, if $\bar{h}'_{s,i}$ fires in round $\tau + 1$, then the number of leading zeros in $h_1(\bar{x})$ is $|\bar{h}_s| - i = 3\log n - i$. Due to the edges from $\bar{h}_s$ to $\bar{z}$, starting round $\tau_1 + 2$ the neurons $\bar{z}$ holds the binary encoding of the number of leading zeros in $h_1(\bar{x})$. □

Let $j = h_2(\bar{x}) = \text{dec}(\bar{h}_b)$ be the bucket to which the input x is mapped to in round τ_1. We first claim that for every bucket $j' \neq j$, the neurons $\bar{b}_{j'}$ do not change their values (from round τ_0 and as long as the input has not changed). Starting at round $\tau_1 + 1$, the neuron a_j – corresponding to bucket j – fires, where for every $j' \neq j$, the neuron $a_{j'}$ is idle. Thus, the comparison neuron $c_{j'}$ does not fire and therefore for every index $j' \neq j$ the value stored in bucket $\bar{b}_j$ does not change.

As for $\bar{b}_j$, let $v_j(\tau)$ be the value stored in bucket $\bar{b}_j$ in round τ. We claim that if the number of leading zeros in $h_2(\bar{x})$, denoted as v_0, is larger than $v_j(\tau_0)$, then for $\tau_2 = \tau_0 + O(\log\log n)$ it holds that $v_j(\tau_2) = v_s$. We note that if $v_0 \leq v_j(\tau_0)$ by Obs. 3 starting round $\tau_1 + 2$ also $\text{dec}(\bar{z}) \leq v_j(\tau_0)$. Setting $\tau > \tau_1 + 2 - \tau_0$, the comparison neuron c_j will not fire and therefore the value stored in $\bar{b}_j$ does not change.

Claim 16. *If $v_0 > v_j(\tau_0)$, then for round $\tau_2 = \tau_0 + O(\log\log n)$ it holds that $v_j(\tau_2) = v_0$. Moreover, the value stored in $\bar{b}_j$ will not change until the next update is presented.*

Proof. The index neuron a_j starts firing by round $\tau_1 + 1$ due to the incoming edges from $\bar{h}_b$. We set the parameter τ such that $\tau > \tau_1 - \tau_0 + 3$. Thus, in round $\tau' = \tau_0 + \tau + 1$ both σ_τ and a_j fires. Since $\tau' > \tau_1 + 2$, by Obs. 3 in round τ' it also holds that $\mathsf{dec}(\bar{z}) = v_0$ and $\mathsf{dec}(\bar{z}) > v_j(\tau_0)$. We conclude that the comparison neuron c_j fires in round $\tau' + 1$. Due to the inhibitor r_i all neurons in $\bar{b}_j$ are idle starting at round $\tau' + 3$, and due to the edges from the neuron c_i^2, in round $\tau' + 4$ the neurons $\bar{b}_j$ will obtain the value v_0 encoded using the neurons $\bar{z}$.

Additionally, due to the inhibitor $r_{i,2}$, starting round $\tau' + 4$, the neurons $\bar{z}$, C and $\bar{x}$ are idle. Therefore, the comparison neuron c_j will not fire until the next input is presented and no additional update will be performed. The claim follows for $\tau_2 = \tau' + 4$. □

The upper bound of Theorem 7 follows by combining Lemma 7, Claim 16 and Steps (7,8).

Lower bound. Finally, we show that the space-bound of Theorem 7 is nearly optimal by using a reduction from communication complexity.

Lemma 8 (Neural Lower Bound). *Any SNN for maintaining a $(1+\epsilon)$ approximation for the number of distinct elements with constant probability requires $\Omega(1/\epsilon^2 + \log n)$ neurons.*

Proof. The lower bound is based on a reduction from communication complexity in the same manner as was shown for the streaming setting. Specifically, for the streaming setting Woodruff [194] showed a lower bound of $\Omega(1/\epsilon^2)$ space for the $(1+\epsilon)$ distinct elements problem with $\delta = O(1)$. This was shown via a reduction from the Gap-Hamming problem in the public-coin two-party model. In our context, we use a similar reduction in order to show a lower bound of $\Omega(1/\epsilon^2)$ on the number of neurons in an SNN network. Let $\mathcal{N}$ be a network for maintaining an $(1+\epsilon)$ estimate for the number of distinct elements with constant probability using s non-input neurons. We use this network to provide a one-way communication complexity protocol between Alice and Bob. Since the random coins are public, both Alice and Bob can compute the network $\mathcal{N}$ (i.e., with the random edge weights). Alice simulates her input items over the network $\mathcal{N}$ by feeding them as input to the network (for a fixed number of rounds). She then sends to Bob the firing states of the non-input neurons in $\mathcal{N}$. This allows Bob to continue with the network simulation by feeding it its input items. The correctness follows by the correctness of the SNN network $\mathcal{N}$ and the communication complexity lower bound.

To show a lower bound of $\Omega(\log n)$ we will use the reduction to the Disjointedness problem in the communication-complexity setting by Alon at el. [9]. This reduction as well works in the two-party model with public-coins, which allows Alice and Bob to compute the same network $\mathcal{N}$ and simulate its operation over their input items in the same manner as above. □

3.4 Median Approximation

Before presenting the neural computation of the approximate median, we describe the neural implementation of the Count-Min Sketch and prove Theorem 8.

3.4.1 A Neural Implementation of Count-Min Sketch

We follow the streaming implementation of Count-Min by [49] described as follows. The algorithm maintains a data structure that consists of $\ell = O(\log 1/\delta)$ hash tables $T_1, \ldots T_\ell$, each with $b = O(1/\epsilon)$ bins, and each table T_i is indexed using a different pairwise independent hash function h_i (i.e., the output domain of h_i is $\{0,1\}^{\log b}$). The operation $\mathrm{inc}(x)$ increases the value in each bin $T_i[h_i(\bar{x})]$ for every $i \in [\ell]$. The $\mathrm{count}(x)$ operation returns the value $\min_{i \in [\ell]} T_i[h_i(\bar{x})]$.

Fact 4 ([49])**.** $\Pr[\mathrm{count}(x) \notin (f(x), f(x)+O(m/b))] \leq 1/2^{\Omega(\ell)}$ *where $f(x)$ is actual frequency of x in the stream of length m.*

Definition 11 (Neural Count-Min Sketch)**.** *Given parameters $\epsilon, \delta \in (0,1)$, a neural Count-Min sketch network $\mathcal{N}_{\epsilon,\delta}$ has an input layer of $\log n + 1$ neurons denoted as $a, x_1, \ldots x_{\log n}$, an output layer of $\log m$ neurons $y_1, \ldots y_{\log m}$, and a set of s auxiliary neurons. The neurons $\bar{x} = (x_1, \ldots x_{\log n})$ encode the binary representation of an element $x \in [n]$ and the neuron a indicates whether this is an inc or count operation, where $a = 1$ indicates an inc operation. For every fixed input value $\bar{x}$ presented at round t and $a = 0$ (i.e., a count operation), let $\mathcal{N}_{\epsilon,\delta}(\bar{x})$ be the value encoded in binary by the output layer $y_1, \ldots y_{\log m}$ in round $t + \tau_{n,m}$. It holds that $\Pr[\mathcal{N}_{\epsilon,\delta}(\bar{x}) \notin (f(x), f(x) + O(\epsilon m'))] \leq \delta$, where $m' \leq m$ is the stream length by round t and $f(x)$ is the current frequency of x.*

We first describe the network construction to support the $\mathrm{inc}(x)$ operation. Then we explain the remaining network details for implementing a $\mathrm{count}(x)$ operation.

Supporting $\mathrm{inc}(x)$ operations. The network contains $\ell = O(\log 1/\delta)$ sub-networks $\mathcal{H}^1_{n,b}, \ldots, \mathcal{H}^\ell_{n,b}$ each implements a pairwise independent hash function $h_i : \{0,1\}^{\log n} \to \{0,1\}^{\log b}$ using Lemma 6. The output vector of each network $\mathcal{H}^i_{n,b}$ is denoted by $\bar{h}_i$ for every $i \in \{1, \ldots, \ell\}$. Every $\bar{h}_i$ has an inhibitory copy $\bar{h}'_i$.

For each sub-networks $\mathcal{H}^i_{n,b}$, and for every value $j \in \{1, \ldots, b\}$, the network contains a counter sub-network that counts the number of items x in the stream that satisfies $h_i(\bar{x}) = j$. Every counter network is implemented by a neural counter network from Fact 1 with time parameter $t = m$. Let $\mathcal{C}_{i,1}, \ldots, \mathcal{C}_{i,b}$ be the neural counter networks corresponding to the i-th hash network $\mathcal{H}^i_{n,b}$. The counter $\mathcal{C}_{i,j}$ is updated based on the values of the output neurons $\bar{h}_i$ as follows. For every counter $\mathcal{C}_{i,j}$ the network contains an index neuron $c_{i,j}$ with input from $\bar{h}_i$ and $\bar{h'}_i$ which fires only if[1] $\mathrm{dec}(\bar{h}_i) = j$. The input to the counter $\mathcal{C}_{i,j}$ denoted as $e_{i,j}$ is an AND gate

[1]For implementation reasons, verifying that $\mathrm{dec}(\bar{h}_i) = j$ requires input from both $\bar{h}_i$ and $\bar{h'}_i$.

of the input neuron a and the index neuron $c_{i,j}$, firing in $\mathrm{inc}(x)$ operations where $h_i(\bar{x}) = j$. To make sure the counter is incremented once per $\mathrm{inc}(x)$ operation, the network contains an inhibitory neuron denoted as $e'_{i,j}$ which has the same incoming edges and weights as $e_{i,j}$, that inhibits the neurons $e_{i,j}$, $c_{i,j}$ and a. This guarantees that $e_{i,j}$ would be active for exactly *one* round per $\mathrm{inc}(x)$ operation.

Supporting $\mathrm{count}(x)$ **operations.** To support a $\mathrm{count}(x)$ operations, for each counter $\mathcal{C}_{i,j}$, the network includes $\log m$ neurons $\bar{s}_{i,j} = s^1_{i,j}, \ldots s^{\log m}_{i,j}$ which hold the value stored in the counter $\mathcal{C}_{i,j}$ such that $h_i(\bar{x}) = j$. Each neuron $s^k_{i,j}$ is an AND gate of the index neuron $c_{i,j}$ and the j-th output neuron of $\mathcal{C}_{i,j}$. In addition, for every $i \in \{1, \ldots, \ell\}$ there are $\log m$ neurons $\bar{g}_i = g_{i,1}, \ldots g_{i,\log m}$ where the j-th neuron $g_{i,j}$ is an OR gate of all the j-th neurons of the vectors $\bar{s}_{i,1}, \ldots, \bar{s}_{i,b}$. As a result, $\bar{g}_i$ encodes the value stored in $h_i(\bar{x})$. Finally, the output value is set to be the *minimum value* of $\mathsf{dec}(\bar{g}_1), \ldots, \mathsf{dec}(\bar{g}_\ell)$ using the minimum computation network of [131].

Correctness. Our goal is to show the proposed network simulates the Count-Min sketch data structure of [49]. Let (x, a) be an input introduced in round τ_0. We start by showing the correctness of an $\mathrm{inc}(x)$ operation (i.e., when $a = 1$). Specifically, we show that the counters $\mathcal{C}_{i,h_i(\bar{x})}$ are incremented by one, and the remaining counters $\mathcal{C}_{i,j}$ for $j \neq h_i(\bar{x})$ are unmodified.

Claim 17. *For every $i \in \{1, \ldots, \ell\}$, the value encoded in the output neurons of the counter $\mathcal{C}_{i,h_i(\bar{x})}$ is increased by one by round $\tau_2 = \tau_0 + O(\log m + \log\log n)$, and for every $j \neq h_i(\bar{x})$ the output of the counter $\mathcal{C}_{i,j}$ is not incremented. The increment to the $\mathcal{C}_{i,h_i(\bar{x})}$ counters occur only once.*

Proof. By Lemma 6, the neurons $\bar{h}_1, \ldots \bar{h}_\ell$ encodes the values $h_1(\bar{x}), \ldots h_\ell(\bar{x})$ by round $\tau_1 = \tau_0 + O(\log\log n)$. Thus, for every $i \in \{1, \ldots, \ell\}$ starting round $\tau_1 + 1$, the index neuron $c_{i,j}$ fires iff $j = h_i(\bar{x})$. Since $a = 1$, starting round τ_0, the AND gate $e_{i,h_i(\bar{x})}$ fires in round $\tau_1 + 2$ (and $e_{i,j}$ are idle for $j \neq h_i(\bar{x})$). Due to the inhibitor copy of $e_{i,j}$, in round $\tau_1 + 3$ the neuron $e_{i,j}$ does not fire. Moreover, since the inhibitor also inhibits a nd $c_{i,j}$, the neuron $e_{i,j}$ is idle until a new inc operation is presented. Hence, the input neuron $e_{i,j}$ of the counter network $\mathcal{C}_{i,j}$ fires exactly once, and therefore the counter is incremented once, as desired. By Fact 1, it holds that the output neurons of $\mathcal{C}_{i,j}$ hold the correct count by round $\tau_2 = \tau_1 + O(\log m)$. □

We proceed with showing the correctness for the $\mathrm{count}(x)$ operation (i.e., when $a = 1$). For every $i \in \{1, \ldots, \ell\}$ and $j \in \{1, \ldots, b\}$, let $x_{i,j}(\tau)$ be the (decimal) value encoded by the output neurons of the counter $\mathcal{C}_{i,j}$ in round τ. We next show that by round $\tau_0 + O(\log m + \log\log n)$, the output neurons $\bar{y}$ of the Count-Min sketch encode the minimum value in $x_{1,h_1(\bar{x})}(\tau_0), \ldots, x_{\ell,h_\ell(\bar{x})}(\tau_0)$.

Claim 18. *The output neurons $\bar{y}$ encode the value $\min_{i\in\{1,\ldots,\ell\}} x_{i,h_i(\bar{x})}(\tau_0)$ by round $\tau_2 = \tau_0 + O(\log m + \log\log n)$.*

Proof. By Lemma 6, the neurons $\bar{h}_1, \dots \bar{h}_\ell$ encodes the values $h_1(\bar{x}), \dots h_\ell(\bar{x})$ by round $\tau_1 = \tau_0 + O(\log\log n)$. In addition, for every $i \in \{1, \dots, \ell\}$, the neuron $c_{i,j}$ fires starting round $\tau_1 + 1$ iff $j = h_i(\bar{x})$. Since $a = 0$, starting round τ_0 the counter $\mathcal{C}_{i,j}$ is not incremented and $c_{i,j}$ is not inhibited by the inhibitor copy of $e_{i,j}$. Thus combined with the persistence assumption of every input, we conclude that the vector $\bar{s}_{i,h_i(\bar{x})}$ encodes $x_{i,h_i(\bar{x})}(\tau_0)$ starting round $\tau_1 + 2$. In addition, for every $j \neq h_i(\bar{x})$ all the neurons of $\bar{s}_{i,j}$ are idle. Therefore, the neurons $\bar{g}_i$ encode $x_{i,h_i(\bar{x})}(\tau_0)$ starting round $\tau_1 + 3$. The claim follows from fact 2. $\square$

The Theorem follows by combining Claim 17, Claim 18 and Fact 4.

3.4.2 Neural Computation of the Approximate Median

In this section, we present our main technically involved algorithmic result for computing an estimate for the median of the data-stream.

Definition 12 (Approximate Median)**.** *Given $\epsilon, \delta \in (0,1)$ and a stream $\mathcal{S} = \{x_1, x_2, \dots x_m\}$ with each $x_i \in [n]$, in the* approximate median *problem, it is required to output an element $x_j \in \mathcal{S}$ whose rank is $m/2 \pm \epsilon m$ with probability at least $1 - \delta$.*

For ease of notation, assume that n is power of 2. Our neural solution is based on the streaming algorithm of [49], that uses $\widetilde{O}(1/\epsilon)$ space. Up to the logarithmic terms, this space-bound is known to be optimal [101].

Fact 5 (Theorem 5 [49])**.** *For every $\epsilon, \delta \in (0,1)$, there exists a randomized streaming algorithm for computing the ϵ-approximate median with probability $1 - \delta$ and $\widetilde{O}(1/\epsilon)$ space.*

We start by providing a high-level exposition of this streaming algorithm, and then explain its implementation in the neural setting. The latter turns out to be quite involved, yet demonstrating the expressive power of SNN networks.

A high-level description of the streaming algorithm. The algorithm is based on applying a binary search over *range queries* which, roughly speaking, compute the frequency of the elements in a given range.

Definition 13 (Range Queries)**.** *Given a data-stream of numbers $\mathcal{S} = \{x_1, \dots, x_m\}$ with each $x_i \in [n]$, a range query receives a range of number $[a,b] \subseteq [1,n]$ and returns the frequency of the items $\{a, a+1, \dots, b\}$ in the stream $\mathcal{S}$.*

To support range queries with small space, the algorithm maintains $\log n$ data structures of Count-Min sketch, for each of the $\log n$ *dyadic intervals* of $[n]$.

Definition 14 (Dyadic Intervals)**.** *The* dyadic intervals *of the set $[n]$ are a collection of* $\log n$ *partitions of n, $\mathcal{I}_1 \dots, \mathcal{I}_{\log n}$ such that*

$$\mathcal{I}_0 = \{\{1\}, \{2\}, \{3\}, \dots, \{n\}\}$$

$$\begin{aligned}
\mathcal{I}_1 &= \{\{1,2\},\{3,4\},\{5,6\},\ldots,\{n-1,n\}\} \\
\mathcal{I}_2 &= \{\{1,2,3,4\},\{5,6,7,8\},\ldots,\{n-3,n-2,n-1,n\}\} \\
&\ldots \\
\mathcal{I}_{\log n} &= \{\{1,2\ldots n\}\}
\end{aligned}$$

Note that every range $[i,j] \subseteq [n]$ can be written as a union of at most $\log n$ sets from the dyadic intervals. Hence, by introducing $\log n$ Count-Min data structures with parameters $\delta' = \log(\log n/\delta)$ and $\epsilon' = \epsilon/\log n$ for dyadic-intervals of $[n]$, we can answer range queries within an additive error of $m \cdot \epsilon$ with probability $1-\delta$. The approximated median is obtained by employing a Binary search over the range queries [1].

Definition 15 (SNN for the Approximate Median Problem)**.** *Given two integers n, m and additional parameters $\epsilon, \delta \in (0,1)$, an approximate-median network $\mathcal{N}_{n,m}$ has an input layer of $n+1$ neurons, an output layer of $\log n$ neurons and a set of s auxiliary neurons. The input neurons are denoted as $(a, x_1, \ldots, x_n)$ where the neuron a indicates whether this is a median query or an insertion operation. When the input layer represents a median query, the neuron a fires and the neurons $x_1, \ldots x_n$ are idle. For every round t, let $\mathcal{S}_t = \{a_1, a_2, \ldots a_t\}$ be the data-stream presented as input to the network by round t. For any median-query presented in round t, by round $t + \tau_{n,m}$ the output layer encodes an element $y \in \mathcal{S}_t$ whose rank in $\mathcal{S}_t$ is $t/2 \pm \epsilon t$ with probability at least $1-\delta$.*

The challenge: The crux of the streaming algorithm is based on a binary search over range queries. A-priori, it is unclear how to implement such a search using a poly-logarithmic number of neurons. Specifically, the (implicit) decision tree that governs the binary search has a linear size. Since the neural network (unlike the streaming algorithm) has to hard-wire the algorithm description, the explicit encoding of the search tree leads to a linear space solution. Our key contribution is in showing a succinct network construction that simulates the binary search of the streaming algorithm using a nearly matching space bound.

Network description. We next provide a description of the network. Recall that the type of the operation is represented by the input neuron a, where $a = 1$ represents a median query. To avoid cumbersome notation, we assume that n is a power of 2.

Supporting insertion operations. In the high level, the network contains 3 parts (1) a set of $\log n$ neurons that encode the inserted element in its binary form, (2) a neural counter that counts the length of the current stream, and (3) $\log n$ Count-Min sketch sub-networks that maintain the frequencies of the $\log n$ dyadic intervals of $[n]$.

1. The n-length input vector $\bar{x}$ is connected to $\log n$ neurons $\bar{x}' = (x'_1, \ldots, x'_{\log n})$ such that

[1]The same algorithm can be applied for any quantile estimation.

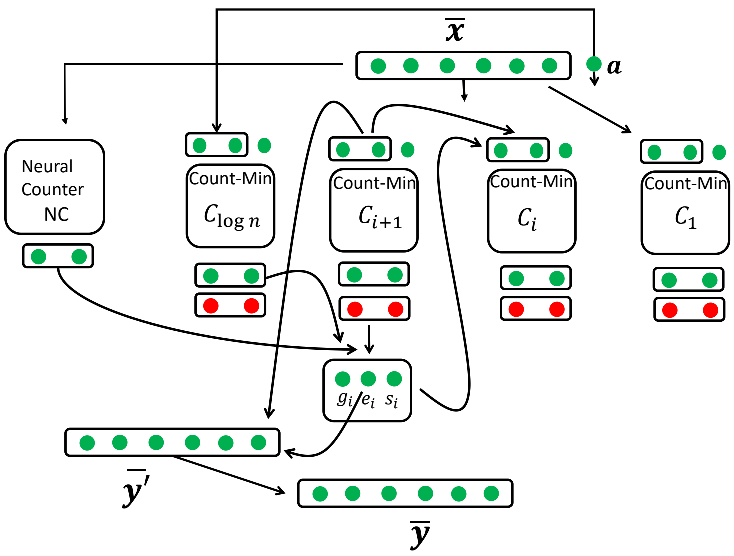

Figure 3.4: A high-level illustration of the approximate median network. The green circles represent excitatory neurons and the red circles represent inhibitory neurons. The input is connected to $\log n$ Count-Min networks, that count the frequencies of the dyadic intervals of $[n]$. On the left, the neural counter module NC counts the total number stream elements. The neurons g_i, s_i, e_i guide the binary search implemented by the network. Once a median is detected, its value is copied to the output neurons using the additional neurons $\bar{y}'_i$.

$\bar{x}'$ encodes the binary representation[1] of the element presented in the input neurons $\bar{x}$. For every $i \in [n]$, $j \in [\log n]$ if the j-th bit in the binary representation of i is equal to 1 then $w(x_i, x'_j) = 1$, and $w(x_i, x'_j) = 0$ otherwise. The bias of every x'_i is set to $b(x'_j) = 1$.

2. The network contains a counter sub-network $\mathcal{NC}_m$ that counts the number of data-items inserted so far. The counter network is implemented by a neural counter network from Fact 1 with time parameter $t = m$. The input neuron to the $\mathcal{NC}_m$ sub-network denoted as a' is an OR gate of the input neurons $\bar{x}$. The output neurons of $\mathcal{NC}_m$ are denoted as $\bar{o} = (o_1, \ldots, o_{\log m})$. Additionally, the network also contain inhibitory copies of the vector $\bar{o}$ denoted by $\bar{o}'$.

   To make sure the counter is incremented once per insertion operation, the network contains an inhibitory copy of a' denoted as r', which inhibits a' and the neurons $\bar{x}$ using large negative weights. As a result, the input neuron a' will be active for exactly *one* round per insertion operation.

3. The network contains $\log n$ sub-networks $\mathcal{C}_1, \ldots \mathcal{C}_{\log n}$ each implements a Count-Min sketch with parameters n, m and $\epsilon' = O(\epsilon / \log n)$, $\delta' = O(\delta / \log n)$ using Theorem 8. For each Count-Min sketch sub-network $\mathcal{C}_i$, let $\bar{z}_i = (z_{i,1}, \ldots z_{i,\log n})$ and b_i be its input

[1]As discussed in the introduction our solution supports both types of input formats: $\log n$-bits of the binary representation or an n-length vector with one active entry.

layer, where the neuron b_i indicates whether the operation is $\mathrm{inc}(x)$ or $\mathrm{count}(x)$. The neuron b_i is an OR gate of the neurons in $\bar{x}$.

For $i \in \{1, \ldots, \log n\}$, the input neurons $\bar{z}_i$ are connected to the binary representation of the input $\bar{x}'$ in the following manner. For every $j \geq i$, the neuron x'_j is connected to the neuron $z_{i,j}$ with large positive weight. For every $j < i$ the neuron $z_{i,j}$ serves as an OR gate of the neurons of $\bar{x}'$. In addition, the neurons b_i, $\bar{z}_i$ are equipped with self-loops. The Count-Min sketch sub-networks are then modified such that these neurons will be inhibited once the computation is complete (by the inhibitory neurons $e'_{i,j}$ of each sub-network respectively).

Supporting median queries. Given a median query, the network computes the approximate median by employing at most $\log n$ steps of binary search. In every step[1] $i \in \{\log n, \ldots, 1\}$, the network obtains a current candidate for the median denoted by χ_i. Initially, $\chi_{\log n} = n/2$. Each χ_i would be provided as input for the i-th Count-Min sketch $\mathcal{C}_i$. The output neurons of $\mathcal{C}_i$ would then define the next candidate χ_{i-1}. Specifically, depending on the rank estimation of χ_i, the network defines the new search range. The width of the search range would be cut by a factor 2 in every step i. Consequently, the algorithm will be using the Count-Min sketch $\mathcal{C}_{i-1}$ which is defined over a partitioning $\mathcal{I}_{i-1}$ where each set is smaller by factor 2 compared to $\mathcal{I}_i$.

1. For every $i \in \{1, \ldots, \log n\}$ the network contains an additional Count-Min sub-network $\mathcal{C}'_i$ which counts the frequencies of the data-elements (similar to $\mathcal{C}_1$). These additional Count-Min sub-networks will be useful in a scenario where for the median item j^*, the frequency of the range $[1, j^*]$ is larger than half, and the frequency of $[1, j^* - 1]$ is too small. This special case would be handled using the $\mathcal{C}'_i$ sub-networks.

   For every sub-network $\mathcal{C}'_i$ with input $\bar{z}'_i, b'_i$, the neuron b'_i serves as an OR gate of the neurons in $\bar{x}$. As for $\bar{z}'_i$, each neuron $z'_{i,j}$ serves as an OR gate of the j-th neuron of $\bar{x}'$ and the j-th neuron of $\bar{z}_i$. Hence, for insertion operations, the sub-network $\mathcal{C}'_i$ is equivalent to $\mathcal{C}_1$. Additionally, the neurons $\bar{z}'_i, b'_i$ are equipped with self-loops. The Count-Min sketch sub-networks are then modified such that these neurons will be inhibited once the computation is complete.

2. For every $i \in \{1, \ldots, \log n\}$ the network contains three *comparison* neurons s_i, g_i, e_i (corresponding to *smaller, greater* or *equal*). These neurons receive their input from the output neurons of the counters $\mathcal{C}_{\log n}, \ldots, \mathcal{C}_i$, $\mathcal{C}'_{\log n}, \ldots, \mathcal{C}'_i$, and the output of the neural counter $\bar{o}, \bar{o}'$. Let $\chi_i = \mathsf{dec}(\bar{z}_i)$ be the median candidate at phase i of the binary-search. The firing states of the comparison neurons are determined as follows. The neuron g_i would fire if frequency estimation of $[1, \chi_i]$ is greater than $m'/2 + \epsilon/2m'$. The neuron

[1]It is convenient to count the steps in a backward manner, as in the i-th step the network will access the i-th Min-Sketch module $\mathcal{C}_i$.

s_i would fire if frequency estimation of $[1, \chi_i]$ is smaller than $m'/2 - \epsilon/2m'$. Finally, e_i would fire if the frequency estimation of $[1, \chi_i]$ is in the range $(m'/2 - \epsilon/2m', m'/2 + \epsilon/2)$.

Denote the output neurons of $\mathcal{C}_i$ by $\bar{f}_i$ and the output neurons of $\mathcal{C}'_i$ by $\bar{f}'_i$. As we will see, the frequency of the range $[1, \chi_i]$ will be decoded by the output neurons of $\mathcal{C}_{\log n}, \ldots, \mathcal{C}_i$ as $\sum_{j=(i-1)}^{\log n} \mathsf{dec}(\bar{f}_j)$. Since the output neurons $\bar{f}_i$ and $\bar{f}'_i$ are *excitatory* (i.e. may only have non-negative outgoing edges), in order for the comparison neurons to fire as desired, for every $i \in [\log n]$ the network introduces inhibitory copies of $\bar{f}_i$ and $\bar{f}'_i$ denoted as $\bar{\phi}_i$ and $\bar{\phi}'_i$ respectively with outgoing edges to the comparison neurons g_i, s_i.

We set the incoming edges of the neuron g_i such that g_i fires if $\sum_{j=i}^{\log n} \mathsf{dec}(\bar{f}_j) - \mathsf{dec}(\bar{f}'_i) > \lfloor \mathsf{dec}(\bar{o})/2 \rfloor + \epsilon/2 \cdot \mathsf{dec}(\bar{o})$. Similarly, the neuron s_i fires if $\sum_{j=i}^{\log n} \mathsf{dec}(\bar{f}_j) - 1 < \lfloor \mathsf{dec}(\bar{o})/2 \rfloor - \epsilon/2 \cdot \mathsf{dec}(\bar{o})$. Regarding the equality neuron e_i, it serves as an AND gate of two intermediate neurons $e_{i,1}$, $e_{i,2}$ such that $e_{i,1}$ fires if $\sum_{j=i}^{\log n} \mathsf{dec}(\bar{f}_j) \geq \lfloor \mathsf{dec}(\bar{o})/2 \rfloor - \epsilon/2 \cdot \mathsf{dec}(\bar{o})$ and $e_{i,2}$ fires if $\sum_{j=i}^{\log n} \mathsf{dec}(\bar{f}_j) - \mathsf{dec}(\bar{f}'_i) + 1 \leq \lfloor \mathsf{dec}(\bar{o})/2 \rfloor + \epsilon/2 \cdot \mathsf{dec}(\bar{o})$.

The neuron g_i is also connected to an inhibitor g'_i which inhibits $\bar{f}_i$ and $\bar{f}'_i$ with large negative weight. The inhibition of $\bar{f}_i$, $\bar{f}'_i$ allows us to maintain the invariant that $\sum_{j=(i-1)}^{\log n} \mathsf{dec}(\bar{f}_j)$ will hold the frequency estimation of $[1, \chi_{i-1}]$ in the next phase when considering χ_{i-1}.

3. The network is augmented with a *timing chain* T which schedules the updates of the neurons $\bar{z}_i$ with the median candidate χ_i. This update should be carefully coordinated to occur only after the neurons g_{i+1}, s_{i+1} and e_{i+1} obtain their values.

   The timing chain T consists of $\tau = \log n \cdot \tau'$ neurons $\sigma_1, \ldots \sigma_\tau$, where $\tau' = \Theta(\log m + \log\log n)$ is an upper bound on the computation time of the Count-Min sub-networks. The first neuron σ_1 has an incoming edge from the input neuron a with weight 1 and bias 1. For $i = 2, \ldots, \tau$, the neuron σ_i has an incoming edge from σ_{i-1} with weight 1 and bias 1. The last neuron σ_τ is an inhibitory neuron, with outgoing edges to the neurons $\bar{z}_1, \ldots, \bar{z}_{\log n}$ and $\bar{z}'_1, \ldots, \bar{z}'_{\log n}$ with large negative weights. The inhibition of these neurons inhibits their self loops in preparation for the next input.

4. In the high level, for $i \in \{\log n, \ldots, 1\}$, every two consecutive sub-networks $\mathcal{C}_{i+1}$ and $\mathcal{C}_i$ are connected in a way that guarantees the following. Let χ_{i+1} be median candidate at phase $i+1$ of the binary search (i.e., that was fed as input to $\mathcal{C}_{i+1}$). Let $\text{freq}([x, y])$ be the estimated frequency of the range $[x, y]$ obtained by the Count-Min sketch networks $\mathcal{C}_{\log n}, \ldots, \mathcal{C}_{i+1}$. Then candidate χ_i is defined as:

$$\chi_i = \begin{cases} \chi_{i+1} - 2^{i-1}, & \text{if } \text{freq}([1, \chi_{i+1}]) > m'/2 + \epsilon/2m' \\ \chi_{i+1} + 2^{i-1}, & \text{if } \text{freq}([1, \chi_{i+1}]) < m'/2 - \epsilon/2m' . \end{cases}$$

   In the remaining case where $\text{freq}([1, \chi_{i+1}]) \in [m'/2 \pm \epsilon/2m']$, the candidate χ_{i+1} will be returned as the output result.

In every step i, the candidate χ_i will be encoded using the input neurons of $\mathcal{C}_i, \mathcal{C}'_i$ denoted as $\bar{z}_i, \bar{z}'_i$. For $i = \log n$, the neurons $\bar{z}_{\log n}$, $\bar{z}'_{\log n}$ have incoming edges from the neuron a such that $\mathsf{dec}(\bar{z}_{\log n}) = 2^{\log n - 1} - 1$[1]. Hence, the first median candidate $\chi_{\log n}$ will be represented in $\bar{z}_{\log n}$ as the binary vector $(0, 0, 1, \ldots, 1)$. For index $i \in [1, \log n - 1]$, toward updating the input neurons $\bar{z}_i$ with the candidate χ_i, the network contains $\log n$ intermediate neurons $\bar{t}_i = t_{i,1}, \ldots, t_{i,\log n}$ with the following connectivity: (i) For $j = 1, \ldots, i-1$, the neuron $t_{i,j}$ has incoming edges from g_{i+1} and s_{i+1} and serves as an OR gate. (ii) For $j = i+2, \ldots \log n$, the neuron $t_{i,j}$ has incoming edges from g_{i+1}, s_{i+1} with weight 1, an incoming edge from $\bar{z}_{i+1,j}$ with weight 2 and bias 3. Hence, if either g_{i+1} or s_{i+1} fired in round τ, the firing state of the j-th neuron $t_{i,j}$ in round $\tau + 1$, is equal to the firing state of $\bar{z}_{i+1,j}$ in round τ. (iii) The neuron s_{i+1} is connected to the neuron $t_{i,i+1}$ with large positive weight.

Next, the incoming edges of the neurons $\bar{z}_i$ are set as follows. Every neuron $z_{i,j}$ has incoming edges from $\bar{t}_i$, and $\sigma_{(\log n - i)\cdot\tau'}$, where we set the weights and bias such that $z_{i,j}$ fires either due to the signal from $\bar{x}'$ (in case of insertion) or both neurons $t_{i,j}$ and $\sigma_{(\log n - i)\tau'}$ fired. The incoming edge from the timing chain T guarantees that when we update $\bar{z}_i$, the computation of the previous candidates χ_{i+1} has been completed.

5. Once a median estimation is found, the output neurons $\bar{y}$ are updated in the following manner. For every $i \in \{1, \ldots, \log n\}$ the network contains a vector of $\log n$ intermediate neurons $\bar{y}'_i$. The neurons $\bar{y}'_i$ are responsible for updating the output neurons $\bar{y}$ when the candidate χ_i is returned as the median estimation. Every neuron $y'_{i,j}$ serves as an AND gate of the equality neuron e_i and $z_{i,j}$. We then connect the neurons $\bar{y}'_1, \ldots \bar{y}'_{\log n - 1}$ to the output neurons $\bar{y}$, where the j-th output neuron $y_{i,j}$ serves as an OR gate of the j-th neurons $y'_{1,j}, \ldots \bar{y}'_{\log n, j}$.

Ensuring the output is a stream element. We modify the neuron s_i to fire also if $\sum_{j=i}^{\log n} \mathsf{dec}(\bar{f}_j) \leq \mathsf{dec}(\bar{o})/2 - \epsilon/2\mathsf{dec}(\bar{o})$ *and* the inhibitory output neurons of $\mathcal{C}'_i$ are idle. This is done using two intermediate neurons (one for each case). In addition, we set the equality neuron e_i to fire only if both $e_{i,1}$, $e_{i,2}$ fire and at least one of the excitatory output neurons of the sub-network $\mathcal{C}'_i$ fires.

Space and time complexity. The update of the Neural Counter $\mathcal{NC}_m$ requires $O(\log m)$ rounds. Each one of the $2\log n$ Count-Min sketch network requires $O(\log\log n + \log m)$ rounds. Hence a median query is computed within $O(\log n(\log\log n + \log m))$ rounds, and an element insertion is complete within $O(\log\log n + \log m)$ rounds.

Regarding the networks size, the network contains $O(\log n)$ Count-Min sketch networks, with parameters n, m and $\epsilon' = O(\epsilon/\log n)$, $\delta' = O(\delta/\log n)$. Hence, each of the Count-Min sketch networks requires $O(\log n/\epsilon \cdot \log\log n \cdot \log m \cdot \log(\log n/\delta) + \log^2(\log n/\delta))$ neurons. The neural counter requires $O(\log m)$ neurons, and the timing chain consists of $O(\log(\log n(\log\log n +$

[1]or $2^{\lfloor \log n \rfloor - 1} - 1$ if n is not a power of 2

$\log m)))$ neurons. Additionally, we introduce $O(\log^2 n)$ intermediate auxiliary neurons. Thus, the approximate median network contains $\widetilde{O}(1/\epsilon)$ auxiliary neurons.

Correctness. We show the proposed network implements the algorithm of [49]. Let x, a be an input presented at round τ_0. We begin with considering insertion operations, where x represents a stream element $i \in [1, n]$ and $a = 0$. In round $\tau_0 + 1$, the input to the neural counter a' fires, as well as the inhibitor r'. Since r' inhibits $\bar{x}$, and a', the counter $\mathcal{NC}_m$ is incremented exactly once.

Due to the incoming edges from the input neurons $\bar{x}$, at round $\tau_0 + 1$ the neurons $\bar{x}'$ fire, representing the binary encoding of i. Additionally the neuron $b_1, \ldots, b_{\log n}, b'_1, \ldots, b'_{\log n}$ which are the input neurons to the networks $\mathcal{C}_1, \ldots, \mathcal{C}_{\log n}, \mathcal{C}'_1, \ldots, \mathcal{C}'_{\log n}$, representing an $\mathrm{inc}(x)$ operation, fire starting round $\tau_0 + 1$. In round $\tau_0 + 2$ the input neurons $\bar{z}_1, \ldots \bar{z}_{\log n}, \bar{z}'_1, \ldots \bar{z}_{\log n}$ receives the signals from $\bar{x}'$ and begin to fire. Due to the self-loops which enable persistence, and the modification to the Count Min networks which inhibits these neurons once the computation is complete, by Theorem 8 each sub-network $\mathcal{C}_i$ performs the operation $\mathrm{count}(\mathsf{dec}(\bar{z}_i))$ by round $\tau_1 = \tau_0 + O(\log m + \log\log n)$.

We now show that every sub-network $\mathcal{C}_i$ maintains an estimation of the frequencies of the dyadic intervals $\mathcal{I}_i$. For a stream element $k \in [n]$, due to the incoming edges from $\bar{x}'$, the neurons $\bar{z}_j$ encodes the value $\lfloor k/2^j \rfloor$ in round $\tau_0 + 2$. Thus, for every two stream elements k_1, k_2 in the interval $[c \cdot 2^i, c \cdot 2^{i+1} - 1]$ for $c \in \mathbb{N}$, the input to the network $\mathcal{C}_i$ is identical and equals $c \cdot 2^i$. On the other hand, if $k_1/2^i \neq k_2/2^i$, the input presented to the sub-network $\mathcal{C}_i$ when inserting k_1 is different than the input presented when inserting k_2. We conclude that the sub-networks $\mathcal{C}_1, \ldots, \mathcal{C}_{\log n}$ implement the neural Count-Min data structure for the dyadic intervals of $[n]$.

We next turn to consider a median query presented at round τ_0. Hence, in round τ_0 the input neurons $\bar{x}$ are idle and the neuron a fires. We first note that because $\mathcal{NC}_m$ is incremented once per stream element, and the persistence of each element is $\Omega(\log m)$, in round τ_0 the outputs of the neural counter $\bar{o}$ and $\bar{o}'$ encodes the size of the stream in round τ_0 denoted as m'.

For every $i = \log n, \ldots, 1$, let χ_i be the value encoded in the neurons $\bar{z}_i$ in round $\tau_0 + i \cdot \tau' + 1$, where τ' is a parameter which upper bounds the computation time of the Count-Min sub-networks. Note that the assignment of the candidates $\chi_{\log n}, \chi_{\log n - 1} \ldots$ is performed in a sequential manner with time intervals of τ' rounds due to the incoming edges from the timing chain T. We consider the candidate χ_i encoded in the firing state of $\bar{z}_i$ at round $\tau_0 + (\log n - i + 1) \cdot \tau' + 1$ as an iteration of a binary search.

Due to the incoming edges from $\bar{z}_{i+1}$ described in Step (4), candidate χ_i (i.e $\mathsf{dec}(\bar{z}_{\log n - i})$) defers from χ_{i+1} in the following manner.

Observation 4. *If $\chi_i \neq 0$, $\chi_{i+1} \neq 0$, in case g_i fires in round $\tau_0 + (\log n - i + 1) \cdot \tau' + 1$ then $\chi_i = \chi_{i+1} - 2^{i-1}$ and in case s_i fires $\chi_i = \chi_{i+1} + 2^{i-1}$.*

Proof. Let $b_{\log n}, \ldots, b_1$ be the binary representation of the candidate χ_{i+1}, represented in the firing state of $\bar{z}_{i+1}$. By the definition of Step (4), the neuron $t_{i+1,i+1}$ is idle in every round, and therefore $b_{i+1} = 0$. Additionally, since $\chi_{i+1} \neq 0$, for every $j < i+1$ the neuron $t_{j,i+1}$ fires starting round $\tau_0 + (\log n - i) \cdot \tau'$, and therefore $b_{i+1} = 1$.

If g_{i+1} fired, according to step (4) the firing state of the neurons $\bar{t}_i$ in round $\tau_0 + i \cdot \tau' + 1$ encode the binary representation $(b_{\log n}, \ldots, b_{i+2}, 0, 0, 1, \ldots, 1) = (b_{\log n}, \ldots, b_{i+2}, b_{i+1}, 0, b_{i-1}, \ldots, b_1)$. Since $b_i = 1$, it follows that $\chi_i = \chi_{i+1} - 2^{i-1}$.

If s_{i+1} fired, the firing state of the neurons $\bar{t}_i$ in round $\tau_0 + i \cdot \tau' + 1$ encodes the binary representation $(b_{\log n}, \ldots, b_{i+2}, 1, 0, 1, \ldots, 1) = (b_{\log n}, \ldots, b_{i+2}, 1, 0, b_{i-1}, \ldots, b_1)$. Since $b_i = 1$ and $b_{i+1} = 0$ we can conclude that $\chi_i = \chi_{i+1} + 2^i - 2^{i-1} = \chi_{i+1} + 2^{i-1}$.

□

We next note that due to the definition of the insertion operation to the networks $\mathcal{C}_1, \ldots, \mathcal{C}_{\log n}$ when considering the candidate χ_i the output neurons of $\mathcal{C}_i$ encode an estimation of the frequency of the interval $[\chi_i - 2^{i-1} + 1, \chi_i]$.

Observation 5. *For every $\mathcal{C}_i$, if $\chi_i \neq 0$ is presented at round t, by round $t + \tau'$ the output neuron $\bar{f}_i$ encode an $(1+\epsilon')$-approximation of the frequencies of the interval $[\chi_i - 2^{i-1} + 1, \chi_i]$ with probability $1 - \delta'$.*

Proof. By the definition of the step (4) for every coordinate $j < i$ when the i-th candidate χ_i is considered, the neuron $z_{i,j}$ fires (equals one). Hence the $i-1$ least significant bits in the binary representation of χ_i are equal to one.

In addition, when considering insertion operations, by the definition of the incoming edges of $\bar{z}_i$, for all elements between $\lfloor \chi_i / 2^{i-1} \rfloor$ and χ_i the input to the network $\mathcal{C}_i$ is identical and equals to χ_i. Hence, by Theorem 8 and the choice of τ', if χ_i is presented at round t, by round $t + \tau'$ the output neuron $\bar{f}_i$ encode an $(1+\epsilon')$-approximation of the frequencies of the interval $[\lfloor \chi_i / 2^{i-1} \rfloor, \chi_i] = [\chi_i - 2^{i-1} + 1, \chi_i]$ with probability $1 - \delta'$ □

Toward proving our search method implements the algorithm of [49], we show that as long as a median estimation has not been found, for every candidate χ_i the output neurons $\bar{f}_{\log n}, \ldots, \bar{f}_i$ encode the frequency of the range $[1, \chi_i]$.

Claim 19. *If the output neurons of the network $\bar{y}$ did not fire by round $\tau_0 + (\log n - i + 1) \cdot \tau' + 4$, in round $\tau_0 + (\log n - i + 1) \cdot \tau'$ it holds that $\mathsf{dec}(\bar{f}_{\log n}) + \cdots + \mathsf{dec}(\bar{f}_i)$ encodes a $(1 + (\log n - i + 1) \cdot \epsilon')$ approximation for the frequency of $[1, \chi_i]$ with probability $1 - (\log n - i + 1) \cdot \delta'$.*

Proof. By induction on i starting $i = \log n$ towards $i = 1$. For $i = \log n$ in round $\tau_0 + 1$ the neurons $\bar{z}_{\log n}$ encode the element $\chi_{\log n} = 2^{\log n - 1} - 1$, and by Obs. 5, the output neurons $\bar{f}_{\log n}$ encodes an $1 + \epsilon'$ approximation of the frequency of the dyadic interval $[1, 2^{\log n - 1} - 1]$ by round $\tau_0 + \tau'$. Assume the claim holds for the $(i+1)$-th candidate, and consider phase i.

For every j let $\tau(j) = \tau_0 + (\log n - j) \cdot \tau'$. Since the output neurons $\bar{y}$ did not fire by round $\tau(i-1)+4$, no equality neuron e_j has fired previously. Thus, $\chi_i \neq 0$ in round $\tau(i)$. Moreover, due to the timing chain T starting round $\tau(i+1)+1$ it holds that $\chi_{i+1} = \mathsf{dec}(\bar{z}_{i+1}) \neq 0$.

Hence, by the induction assumption it holds that $f_1 = \mathsf{dec}(\bar{f}_{\log n-1}) + \cdots + \mathsf{dec}(\bar{f}_{i+1})$ encodes an $(1 + (\log n - i) \cdot \epsilon')$ approximation of the frequency of $[1, \chi_{i+1}]$ by round $\tau(i)$ with probability $1 - (\log n - i) \cdot \delta'$. Since $\chi_i \neq 0$ it holds that in round $\tau(i)+1$ either g_{i+1} fired, or s_{i+1} fired.

- If s_{i+1} fired, then by Obs. 4 it holds that $\chi_i = \chi_{i+1} + 2^{i-1}$. When we query $\mathcal{C}_i$ by Claim 5 in round $\tau_0 + (\log n - i + 1)\tau'$ it holds that $\mathsf{dec}(\bar{f}_i)$ holds an $(1+\epsilon')$ approximation for the frequency of $[\chi_i - 2^{i-1} + 1, \chi_i] = [\chi_{i-1} + 1, \chi_i]$ with probability $1 - \delta'$. Thus, we conclude that $\mathsf{dec}(\bar{f}_{\log n}) + \cdots + \mathsf{dec}(\bar{f}_{\log n - i}) = f_1 + \mathsf{dec}(\bar{f}_{\log n - i})$ is a $(1 + (\log n - i + 1)\epsilon')$-approximation of the frequency $[1, \chi_i]$ with probability $1 - (\log n - i + 1) \cdot \delta'$.

- If $g_{\log n - i + 1}$ fired, by Obs. 4 it holds that $\chi_i = \chi_{i+1} - 2^{i-1}$. Recall that by step (2), g'_{i+1} inhibits the neurons $\bar{f}_{i+1}$ starting round $\tau_0 + (\log n - i + 1)\tau' - 1$ [1]. By Obs. 5 the neurons $\bar{f}_{i+1}$ holds an approximation of the frequency of of the interval $[\chi_{i+1} - 2^i + 1, \chi_{i+1}$. Thus, in round $\tau_0 + (\log n - i + 1)\tau'$ it holds that $\mathsf{dec}(\bar{f}_{\log n - 1}) + \cdots + \mathsf{dec}(\bar{f}_i) = f_1 - \mathsf{dec}(f_{i+1}) + \mathsf{dec}(\bar{f}_i)$ is an $(1 + i \cdot \epsilon')$-approximation of the frequency of

  $$[1, \chi_{i+1} - 2^i] + [\chi_i - 2^{i-1} + 1, \chi_i] = [1, (\chi_i + 2^{i-1}) - 2^i] + [\chi_i - 2^{i-1} + 1, \chi_i] = [1, \chi_i]$$

  with probability $1 - (\log n - i + 1)\delta$.

$\square$

Combining Claim 19 with Steps (2) and (4) of the network description we conclude that in every iteration i, either we find a median estimation due to the equality neuron e_i, or our candidate χ_i is too small and we increase the next candidate by $n/2^i$, or our candidate is too large and we decrease it by $n/2^i$. The proof of Theorem 9 then follows from the choice of ϵ', δ' and Fact 5.

3.5 Streaming Lower Bounds Yield SNN Lower Bounds

We conclude by addressing Question 2, giving a generic reduction that lets us simulate a space-efficient SNN with a space-efficient neural network. This establishes a tight connection between the two models – any streaming space lower bound yields a near-matching neural-space lower bound.

Complexity classes in the SNN model. For integer parameters n, m, S, let $\mathcal{SNN}_{\det}(n, m, S)$ be the set of all data-stream problems $P_{n,m}$ defined over universe $[n]$ and stream length at most m that are solvable by a deterministic SNN with (i) at most $O(S)$ non-input neurons (i.e.,

[1]The parameter τ' is chosen to be large enough for that purpose

auxiliary and output neurons) and (ii) polynomially bounded edge weights (by n and m). Let $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n,m,S)$ be the class of all data-stream problems $P_{n,m}$ in $\mathcal{SNN}_{\text{det}}(n,m,S)$ whose network solution also have in addition a polynomial persistence time (in n and m). That is, the problems in $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n,m,S)$ are solvable in polynomial-time by a deterministic SNN that has properties (i,ii).

We also consider the class of data-stream problems that are solvable by a randomized SNN. Let $\mathcal{SNN}_{\text{rand}}(n,m,S,\delta)$ be the set of all data-stream problems $P_{n,m}$ that are solvable by a randomized SNN with: (i) at most $O(S)$ non-input neurons, (ii) polynomially bounded edge weights, and (iii) $\leq \delta$ failure probability on any input. The class $\mathcal{SNN}^{\text{poly}}_{\text{rand}}(n,m,S,\delta)$ is a sub-class of $\mathcal{SNN}_{\text{rand}}(n,m,S,\delta)$ that requires also a polynomial persistence time.

Complexity classes in the streaming model. Let $\mathcal{ST}_{\text{det}}(n,m,S)$ be the class of all data-stream problems for which there exists a single-pass deterministic streaming algorithm for the problem using space $O(S)$ (potentially with exponentially large update time). Also, let $\mathcal{ST}_{\text{rand}}(n,m,S,\delta)$ be the class of all data-stream problems for which there exists a single-pass randomized streaming algorithm that solves the problem with failure probability $\leq \delta$ using space $O(S)$. One can also define the classes $\mathcal{ST}^{\text{poly}}_{\text{det}}(n,m,S)$ and $\mathcal{ST}^{\text{poly}}_{\text{rand}}(n,m,S,\delta)$ which require polynomial update time.

We start by showing that any deterministic SNN with space S for a given data-stream problem $P_{n,m}$ yields an S-space deterministic streaming algorithm for the problem.

Lemma 9. *For every n,m,S, we have:*
$\mathcal{SNN}_{\text{det}}(n,m,S) \subseteq \mathcal{ST}_{\text{det}}(n,m,S)$ *and* $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n,m,S) \subseteq \mathcal{ST}^{\text{poly}}_{\text{det}}(n,m,S)$.

Proof. Fix the parameters n,m,S, and consider a problem $\Pi \in \mathcal{SNN}_{\text{det}}(n,m,S)$. Let $\mathcal{N}$ be the SNN for the problem Π. Thus $\mathcal{N}$ has S auxiliary and output neurons. We now describe a streaming algorithm for Π that uses space S. The algorithm traverses the stream and feeds each item as an input to the network $\mathcal{N}$ (with sufficient large persistence time). Importantly, when considering the subsequent input item, the streaming algorithm only keeps the current firing states of the S auxiliary and output neurons. The correctness follows immediately by the correctness of the network $\mathcal{N}$. The space complexity is S bits corresponding to the firing states of the (non-input) neurons in $\mathcal{N}$. The proof that $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n,m,S) \subseteq \mathcal{ST}^{\text{poly}}_{\text{det}}(n,m,S)$ is analogous since the update time of the streaming algorithm is polynomial in the network size and the persistence time of the network. $\square$

Pseudorandomness for neural networks. Our next goal is to simulate space-efficient randomized SNNs for data-stream problems with small-efficient streaming algorithms. The main barrier arises in the case where the edge weights of the network $\mathcal{N}$ are chosen randomly according to some distribution. Since an S-space network with n input neurons might have $\Omega(Sn+S^2)$ edges, the explicit specification of the edge weights is too costly for our purposes.

To overcome this barrier, we will use pseudorandom generators [188].

Definition 16 (PRG)**.** *A deterministic function $G : \{0,1\}^d \to \{0,1\}^n$ for $d < n$ is a (t,ϵ) pseudorandom generator (PRG) if any circuit C of size at most t distinguishes a uniform random string $U \leftarrow \{0,1\}^n$ from $G(R)$, where $R \leftarrow \{0,1\}^d$, with advantage at most ϵ. The parameter d is called the* seed length.

Proposition 1 (Prop. 7.8 in [188])**.** *For all $n \in \mathbb{N}$ and $\epsilon > 0$, there exists a (non-explicit) (n,ϵ) pseudorandom generator (PRG) $G : \{0,1\}^d \to \{0,1\}^n$ with seed length $d = O(\log n + \log 1/\epsilon)$.*

The existence of the PRG from Prop. 1 is shown via the probabilistic method. Such a PRG can be found in a brute-force manner, by iterating over all n-size circuits and all functions $G : \{0,1\}^d \to \{0,1\}^n$ in some fixed order. The desired function G^* is the *first* function that fools the family of all n-size circuits.

Since an SNN with n input neurons, S non-input neurons for $S = \mathsf{poly}(n)$, and polynomial persistence time can be computed in polynomial time (and thus also by a circuit of polynomial size), we have the following:

Lemma 10. *Any SNN $\mathcal{N}$ with n input neurons, S non-input neurons for $S = \mathsf{poly}(n)$, and persistence time $\mathsf{poly}(n)$ in an m-length stream can be simulated using a total space of $O(S + \log(nm))$. The success guarantee of the simulation is $1 - 1/\mathsf{poly}(n,m)$.*

Proof. Consider a (centralized, offline) algorithm that given an ordered stream of length $m' \leq m$ of elements in $[1,n]$ evaluates the output of the network $\mathcal{N}$ on that stream. This algorithm can be implemented in time $\mathsf{poly}(n,m)$ and thus there exists a circuit of size $M = \mathsf{poly}(n,m)$ that implements this algorithm. Our goal is to simulate this circuit using a random seed of length $d = O(\log(nm))$ while reducing the success guarantee by an additive term of $1/\mathsf{poly}(n,m)$. To do that, we will use the PRG construction of Prop. 1 that given a random seed of size d fools the family of all circuits of size at most M with probability $1 - 1/\mathsf{poly}(M)$.

We assume that the PRG function G is hard-coded in the streaming algorithm in the following sense. There is a PRG oracle that given a d length seed R and an index i outputs the i'th bit of $G(R)$. We can think of the code of G as simply comprising a look up table, but note that this code is not part of the space complexity of the streaming algorithm, which only includes data written by the algorithm while processing the stream. The seed R must be chosen randomly at the beginning of the stream and then stored, and thus is included in the space complexity. We also note that the evaluation time G (i.e., outputting each bit) might be exponential.

We now describe how to simulate $\mathcal{N}$ using $O(\log(nm)+S)$ space using this oracle. We store the seed of $O(\log(nm))$ random bits R and the current firing states of all non-input neurons in $\mathcal{N}$. Then, as we traverse the stream, for every data-item in the stream, the algorithm evaluates the network $\mathcal{N}$ on that data-item using the PRG oracle in the following manner. The simulation works in a round by round and a neuron by neuron fashion which only stores $O(\log nm)$ bits from $G(R)$ at any given time. To evaluate the firing state of neuron u in layer $i \geq 0$, the total incoming edge weight of u is computed as follows. Let $v_1, \ldots, v_k$ be the incoming neighbors of u. The firing states in round $i-1$ are stored explicitly (this is indeed within the space bound S). For each v_j that fired in round $i-1$ we look up $O(\log nm)$ entries in $G(R)$ which describe the edge weight $w(v_i, u)$. We note that, without loss of generality we can assume that the edge weights have precision $1/\mathsf{poly}(n,m)$ and thus can be described with $O(\log nm)$ bits. Rounding any edge weights to have this precision will not affect the success probability of the network by more than a $1/\mathsf{poly}(n,m)$ factor. We accumulate the edge weights into a value P, the total incoming potential of u, which again requires $O(\log nm)$ bits to store. Finally, using P we evaluate the probability that u fires in round i. We again can round this probability to $1/\mathsf{poly}(n,m)$ precision, and thus by looking up $O(\log nm)$ entries in $G(R)$ evaluate if u fires in round i. We proceed in this way, iterating over all S non-input neurons and storing their states in round i, before proceeding to the next round. Overall, our space complexity remains bounded by $O(\log(nm) + S)$.

The success probability of the algorithm overall is decreased by an additive $1/\mathsf{poly}(n,m)$ term, due to the rounding of edge weights and probabilities and the use of pseudorandom rather than truly random bits. □

Lemma 10 implies that any randomized SNN with space S that solves a streaming problem $P_{n,m}$ with probability $1-\delta$ in polynomial time translates into a randomized streaming algorithm for $P_{n,m}$ using space of $S + O(\log(nm))$. We therefore have:

Theorem 11. $\mathcal{SNN}^{\text{poly}}_{\text{rand}}(n,m,S,\delta) \subseteq \mathcal{ST}_{\text{rand}}(n,m,S+O(\log(nm)),\delta+1/\mathsf{poly}(n,m))$.

A useful implication of Theorem 11 is that any space lower bound in the streaming model immediately translates into space lower bound for networks that have a polynomial persistence time on the input stream.

Corollary 2. *Let $P_{n,m}$ be a data-stream problem for which any randomized streaming algorithm that solves the problem with probability $1-\delta$ requires space $\Omega(S(n,m,\delta))$. Then, any SNN for solving $P_{n,m}$ within polynomial number of rounds with probability at least $1-\delta+1/\mathsf{poly}(n,m)$ requires space of $\Omega(S(n,m,\delta)-\log(nm))$.*

Proof. Assume towards contradiction that there is an SNN for solving $P_{n,m}$ with probability at least $1-\delta-\mathsf{poly}(1/m)$ within polynomial number of rounds, and using space of $o(S(n)+\log m)$. Thus, $P_{n,m} \in \mathcal{SNN}^{\text{poly}}_{\text{rand}}(n,m,o(S(n,\delta)-\log m),1-\delta+\mathsf{poly}(1/m))$. The exact specification of the $\mathsf{poly}(\cdot)$ terms are given by Theorem 11. By Theorem 11, it then holds that $P_{n,m} \in \mathcal{ST}_{\text{rand}}(n,m,o(S(n,\delta)),1-\delta)$. Contradiction for the fact that solving $P_{n,m}$ with probability $1-\delta$ requires streaming space of $\Omega(S(n,\delta))$. The corollary follows. □

PART II

RESILIENT DISTRIBUTED ALGORITHMS

4 Background and Preliminaries

As networks expand in complexity and scale, they become more vulnerable to various faults, ranging from hardware failures to adversarial attacks. As part of this thesis, we study resilient distributed algorithms designed to maintain their correctness even when some of the edges within the network are corrupted. We focus on an adversary with unbounded computational power, also known as *byzantine*, who can deviate from the protocol and act arbitrarily. We consider the full-information model, where the adversary is allowed to see the entire graph, the messages sent throughout the algorithm, and the internal randomness of the vertices.

Among the most fundamental problems studied in resilient computation are byzantine agreement and byzantine broadcast [159, 109, 57]. These problems are central to a wide range of resilient algorithms and cryptographic multiparty computation protocols e.g, [21, 79, 41]. In the byzantine broadcast problem, the goal is for a designated source vertex to send some message m to all the vertices in the network. The objective is for all reliable vertices to output the same message, and if the source is reliable then the output message should be m. For the byzantine agreement problem, each vertex holds an input value, and the goal is for all reliable vertices to agree on one of these inputs. We note that in terms of feasibility, both problems are equivalent and can be reduced from each other.

Most previous work concentrate on solving these problems in complete networks with adversarial vertices e.g [75, 58, 92, 46, 69, 150, 184, 63, 103]. In terms of feasibility, Fischer, Lynch, and Merritt [67], showed that solving the byzantine agreement and byzantine broadcast problems is impossible if the adversary controls $f > n/3$ of the vertices in the graph, where n

is the total number of vertices. Fischer and Lynch [66] also gave a lower bound of $(f+1)$ on the number of rounds of any deterministic agreement or broadcast algorithm. On the positive side, the classic protocol of Pease, Shostak and Lamport [159] has optimal resilience of $f = n/3$ and optimal worst case of $(f + 1)$ rounds. However, the message complexity of their protocol is exponentially large. Garay and Moses [75] improved the exponential message complexity, presenting a byzantine agreement protocol with optimal resilience, optimal worst case $(f + 1)$ rounds, and polynomial message size.

Turning to general graphs topologies, the approaches used for complete networks that are applicable in arbitrary networks either require large messages or polynomial number of rounds even for a single fault. In its seminal work, Dolev [57] showed that any given graph can tolerate up to f adversarial vertices iff the graph is $(2f + 1)$ vertex-connected. The broadcast protocol introduced by Dolev assumes the topology of the graph is known to the vertices, and uses communication over $(2f + 1)$ vertex disjoint paths. We note that computing such disjoint paths efficiently when the topology is unknown is a challenging task independently. In recent years, Chlebus, Kowalski, and Olkowski [43] extends the algorithm of Garay and Moses [75] to general $(2f + 1)$ vertex-connected graphs with minimum degree $3f$, using exponentially large messages.

As for adversarial edge faults, Pelc [160] studied adversarial edges in general graph topologies, stating that the byzantine agreement and broadcast problems with f adversarial edges can be solved iff the graph is $(2f + 1)$ edge-connected. The algorithm of [160] uses exponentially large messages, which can be reduced to polynomial messages if the graph's topology is known. Barak at el. [18] considered complete graphs in which the adversary controls *all* the edges in the network. Given several cryptographic assumptions, [18] presented a protocol that limits the adversary's power to simulate subsets of the vertices in a consistent manner.

Other studies consider *mobile* edge faults, in which the adversary can control a different set of edges in every round. In this model, the studies on agreement and broadcast mainly focus on complete graphs, and both possibility and impossibility results have been established [169, 170]. For general graphs, Santoro and Widmayer [171] bounded the number of faults and stated the necessary topological requirements for solving any non-trivial form of agreement. They also showed that these bounds are tight provided that the graph's topology is known. Another model considering edge faults is the *hybrid model*, in which the network contains both vertex and mobile edge faults. This model was mainly studied in complete graphs, with the assumption that the number of edge faults touching each vertex is limited [173, 174, 179, 27, 26].

Our approach. A common theme in this literature concerns the feasibility of the computation, e.g., characterizing the requirements a network should satisfy for eventually reaching an agreement in the presence of byzantine faults. The efficiency aspects, in terms of the number of rounds in bandwidth restricted models, have received less attention. We therefore ask, what is the cost in terms of number of rounds of resilient computation in arbitrary graph topologies with bandwidth limitations. We consider the classical CONGEST model [164], in which execution proceeds in synchronous rounds and in each round, every vertex can send a message of size $O(\log n)$ to each of its neighbors. This constraint realistically captures scenarios where bandwidth is limited, making it relevant for various real-world applications such as sensor networks, peer-to-peer systems, and routing in the Internet. Additionally, we assume that each vertex only holds a local view of the network. That is, initially every vertex only knows the identifiers of its neighbors, and a polynomial estimal on the number of vertices n.

The study of resilient algorithms in the CONGEST model was initiated by Parter and Yogev [155, 156, 158]. Motivated by various applications for resilient distributed computing, Parter and Yogev introduced the notion of *low-congestion cycle covers* as a basic communication backbone for reliable communication [155]. At the high level, a low-congestion cycle cover is a collection of cycles covering all the edges in the graph, where all cycles are both short and nearly edge-disjoint. [155] proved the existence of cycle covers in bridgeless graphs, and demonstrate their usefulness in resilient computation. Given low-congestion cycle covers, they obtain a simulation methodology, that can take any distributed algorithm and *compile* it into an equivalent algorithm that is resilient against a single adversarial edge[1]. Their compilation protocol assumes a fault free preprocessing phase, in which the cycle covers are computed. Alternatively, if the topology of the graph is known as assumed in many previous works, then there is no need for the preprocessing phase.

In this thesis we cover work originally published in [87, 88]. In Chapter 5 [87] we present broadcast algorithms in the CONGEST model that are resilient against adversarial edges. In Chapter 6 we extend the work of [155, 156] and show a general simulation methodology that coverts any (fault-free) CONGEST algorithm into an equivalent algorithm that is resilient against adversarial edges. Our compilation methodology does not require preprocessing in a fault-free setting, and also handles multiple adversarial edges. A very recent work by Fischer and Parter [68] extended our simulation methodology (and that of [155]) to handle mobile edge faults. Their resilient algorithms require a fault-free reprocessing phase, and obtain round complexity that nearly matches the round complexity obtained for static adversaries, presented in Chapter 6.

In the remainder of this chapter we define the computational model known as the *adversarial* CONGEST *model* and give some basic tools used in the following chapters.

Preliminaries and Basic Tools

We start with formally defining the computational model we consider in this work.

The Adversarial CONGEST model: The network is abstracted as an n-vertex graph $G =$

[1]The algorithm of [155] also handles mobile edge faults, in which the adversary can control a different edge in every round.

(V, E), with one processor on each vertex. Each vertex has a unique identifier of $O(\log n)$ bits. Initially, the processors only know the identifiers of their incident edges[1], as well as a polynomial estimate on the number of vertices n.

There is a computationally unbounded adversary that controls a fixed set of edges F^* in the graph. The set of F^* edges are denoted as *adversarial*, and the remaining edges $E \setminus F^*$ are denoted as *reliable*. The vertices do not know the identity of the adversarial edges in F^*, but they do know the bound f on the cardinality of F^*. We consider the full information model where the adversary knows the graph, the messages sent through the graph edges in each round, and the internal randomness and the input of the vertices. On each round, the adversary can send $O(\log n)$ bits along each of the edges in F^*. The adversary is adaptive as it can determine its behavior in round r based on the overall communication up to round r.

The primary complexity measure of this model is the *round* complexity. In contrast to many prior works in the adversarial setting, in our model, the vertices are *not* assumed to know the graph's topology, and not even its *diameter*.

Basic tool: Covering families. Our distributed algorithms in the adversarial CONGEST model are based on communication over a collection of G-subgraphs that we denote as covering family. These families are used extensively in the context of fault-tolerant network design [10, 193, 56, 81, 152, 158, 38, 31, 102].

Definition 17 ((L, k) covering family). *For a given graph G, a family of G-subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ is an (L, k)* covering family *if for every $\langle u, v, E' \rangle \in V \times V \times E^{\leq k}$ and any u-v path $P \subseteq G \setminus E'$ of length at most L, there exists a subgraph G_i such that (P1) $P \subseteq G_i$ and (P2) $E' \cap G_i = \emptyset$.*

As the graph topology is unknown, one cannot hope to compute a family of subgraphs that are completely known to the vertices. Instead, we require the vertices to *locally* know the covering family in the following manner.

Definition 18 (Local Knowledge of a Subgraph Family). *A family of ordered subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ where each $G_i \subseteq G$, is* locally known *if given the identifier of an edge $e = (u, v)$ and an index i, the vertices u, v can locally determine if $e \in G_i$.*

By Karthik and Parter [102], we have the following:

Fact 6 ([102]). *Given a graph G and integer parameters L and k, there exists a 0-round algorithm that allows all vertices to locally know an (L, k) covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ such that $\ell = ((Lk \log n)^{k+1})$.*

Proof. The proof follows by using the construction of (L, k) Replacement Path Cover (RPC) $\mathcal{G}$ by [102]. Assuming that vertex IDs are in $[1, \mathsf{poly}(n)]$, each vertex can employ this construction

[1]This is known as the standard KT1 model [14].

locally and compute a collection of subgraphs $\mathcal{G}' = \{G'_1, \ldots, G'_k\}$ where each G'_i is a subgraph of the complete graph G^* on all vertices with IDs in $[1, \mathsf{poly}(n)]$. The set $\mathcal{G} = \{G_1, \ldots, G_k\}$ is defined by $G_i = E(G'_i) \cap E(G)$. We next claim that since $\mathcal{G}'$ is an (L, k) RPC for G^* it also holds that $\mathcal{G}$ is an (L, k) RPC for G. To see this, consider any L-length u-v path P in $G \setminus \{F\}$ for some set $F \subseteq E$ of size at most k. Clearly both F and P are in G^*. Therefore there exists a subgraph G'_i containing P and avoiding F. It then holds that $G_i = G \cap G'_i$ contains P and avoids F as well.

The family of graphs $\mathcal{G}$ is locally known since in the construction of [102], each subgraph is identified with a hash function $h : [m] \to [q]$, in some family of hash functions $\mathcal{H}$, and an index $i \in [q]$. The subgraphs to which an edge e belongs depend only on the value of $h(ID(e))$. Since the family of hash functions $\mathcal{H}$ can be locally computed by each vertex, we have that given an edge identifier, every vertex can locally compute the indices of the subgraphs in $\mathcal{G}$ to which e belongs. Note that since the vertices do not know G, they also do not know the graph G_i but rather the corresponding graph G'_i, and thus we only require them to know the index i of the subgraph. □

In the context of $(2f + 1)$ edge-connected graphs with f adversarial edges, we set $L = O(fD)$ and $k = O(f)$. We will use the following observation.

Observation 6. *Consider an n-vertex D-diameter graph $G = (V, E)$ and let u, v be a pair of vertices that are connected in $G \setminus F$ for some $F \subseteq E$. It then holds that* $\text{dist}_{G \setminus F}(u, v) \leq 2(|F| + 1) \cdot D + |F|$.

Proof. Let T be a BFS tree in G rooted at some source s. The forest $T \setminus F$ contains at most $|F| + 1$ trees of diameter $2D$. Then, the u-v shortest path P in $G \setminus F$ can be transformed into a path P' containing at most $|F|$ edges of P, as well as, $|F| + 1$ tree subpaths of the forest $T \setminus F$. Therefore, $|P'| \leq 2(|F| + 1) \cdot D + |F|$ as desired. □

Our algorithm makes use of the following definition for a minimum s-v cut defined over a collection of s-v *paths*.

Definition 19 (Minimum (Edge) Cut of a Path Collection)**.** *Given a collection of s-v paths $\mathcal{P}$, the minimum s-v cut in $\mathcal{P}$, denoted as MinCut$(s, v, \mathcal{P})$, is the minimal number of edges appearing on all the paths in $\mathcal{P}$. I.e., letting MinCut$(s, v, \mathcal{P}) = x$ implies that there exists a collection of x edges E' such that for every path $P \in \mathcal{P}$, it holds that $E' \cap P \neq \emptyset$.*

Notations. Throughout, the diameter of the given graph G is denoted by D, and the number of vertices by n. For a graph $G = (V, E)$, a subgraph $G' \subseteq G$, and vertices $u, v \in V(G')$, let $\pi(u, v, G')$ be the unique u-v shortest path in G' where shortest-path ties are decided arbitrarily in a consistent manner. Let $N(u, G)$ be the neighbors of vertex u in the graph G. When the graph G is clear from the context we may omit it and write $N(u)$. For a path $P = [u_1, \ldots, u_k]$ and an edge $e = (u_k, v)$, let $P \circ e$ denote the path obtained by concatenating e to P. Similarly, for two paths $P_1 = [u_1, \ldots, u_k], P_2 = [u_k, u_{k+1}, \ldots, u_\ell]$ denote the concatenated path $[u_1, \ldots, u_k, u_{k+1}, \ldots, u_\ell]$ by $P_1 \circ P_2$. Given a path $P = [u_1, \ldots, u_k]$ denote the sub-path from u_i to u_ℓ by $P[u_i, u_\ell]$. The term $\widetilde{O}(\cdot)$ hides $\mathsf{poly}(\log n)$ factors, and the term $\widehat{O}(\cdot)$ hides $2^{O(\sqrt{\log n})}$factors[1].

[1]The latter factors arise by the (fault-free) distributed computation of cycle covers by [156].

5 Broadcast CONGEST Algorithms against Adversarial Edges

5.1 Introduction

Guaranteeing the uninterrupted operation of communication networks is a significant objective in network algorithms. The area of resilient distributed computation has been receiving a growing attention over the last years as computer networks grow in size and become more vulnerable to byzantine failures. Since the introduction of this setting by Pease Shostak and Lamport [159, 109], distributed broadcast algorithms against various adversarial models have been studied in theory and practice for over more than four decades. Resilient distributed algorithms have been provided for broadcast and consensus [57, 58, 65, 34, 184, 169, 25, 170, 24, 63, 75, 69, 107, 162, 103, 59, 137, 92, 47, 105], as well as for the related fundamental problems of gossiping [30, 16, 37], and agreement [58, 159, 33, 46, 75]. See [161] for a survey on this topic. A key limitation of many of these algorithms is that they assume that the communication graph is the complete graph.

In this work, we concentrate with communication graphs of arbitrary topologies. In particular, we addresses the following basic question, which is still fairly open, especially in the CONGEST model of distributed computing [164]:

Question 3. *What is the cost (in terms of the number of rounds) for providing resilience against* *adversarial edges in distributed networks with arbitrary topologies?*

An important milestone in this regard was made by Dolev [57] who showed that the $(2f+1)$ vertex-connectivity of the graph is a necessary condition for guaranteeing the correctness of the computation in the presence of f adversarial vertices. Pelc [160] provided the analogous argument for f adversarial edges, requiring edge-connectivity of $(2f+1)$. The broadcast protocol presented therein requires a linear number of rounds (linear in the number of vertices) and exponentially large messages. Byzantine broadcast algorithms for general graph topologies have been addressed mostly under simplified settings [162], e.g., probabilistic faulty models [163, 160], cryptographic assumptions [73, 2, 1, 145], or under bandwidth-*free* settings (e.g., allowing neighbors to exchange exponentially large messages) [57, 137, 105, 107, 59, 105, 43].

We consider the adversarial CONGEST model which extends the standard CONGEST model, described in Chapter 4. To address Question 3, we provide a comprehensive study of the adversarial broadcast problem, which is formally defined as follows:

The adversarial broadcast task: Given is a $(2f+1)$ edge-connected graph $G=(V,E)$ and a set $F \subset E$ of $|F| \leq f$ edges controlled by the adversary. There is a designated source vertex $s \in V$ that holds a message m_0. It is then required for all the vertices to output m_0, while ignoring all other messages.

To this date, all existing broadcast algorithms in the adversarial CONGEST model require a polynomial number of rounds, even when handling a single adversarial edge! Recently, Chlebus, Kowalski, and Olkowski [43] extended the result of Garay and Moses [75] to general $(2f+1)$ vertex-connected graphs with minimum degree $3f$. Their algorithms, however, use *exponentially* large communication. Their message size can be improved to polynomial only when using authentication schemes (which we totally avoid in this work). It is also noteworthy that the existing protocols for *vertex* failures might still require polynomially many rounds for general graphs, even for a single adversarial edge and for *small* diameter graphs.

A natural approach for broadcasting a message m_0 in the presence of f adversarial edges is to route the message along $(2f+1)$ edge-disjoint paths between the source vertex s, and each target vertex v. This allows each vertex to deduce m_0 by taking the majority message. This approach has been applied in previous broadcast algorithms (e.g., [57, 160]) under the assumption that the vertices know *the entire graph*, and therefore can compute these edge-disjoint paths. In Chapter 6 we demonstrated that there are D-diameter $(2f+1)$ edge-connected graphs, for which the maximal length of any collection of $(2f+1)$ edge-disjoint paths between a given pair of vertices might be as large as $(D/f)^{\Theta(f)}$. For $f=1$, the length lower bound becomes $\Omega(D^3)$. Providing round efficient algorithms in the adversarial CONGEST model calls for a new approach.

Our approach in a nutshell. Our approach is based on combining the perspectives of fault-tolerant (FT) network design, and distributed graph algorithms. The combined power of these points of view allows us to characterize the round complexity of the adversarial broadcast task as a function of the graph diameter D, and the number of adversarial edges f. This is in contrast to prior algorithms that obtain a polynomial round complexity (in the number of vertices). On a high level, our distributed algorithms are based on propagating messages over a *covering family* of G-subgraphs $\mathcal{G}$, as defined in Chapter 4 (Definition 17).

One can show that for $L = O(fD)$ and $k = O(f)$, by the properties of the covering family, exchanging the message m_0 over all subgraphs in $\mathcal{G}$ (in the adversarial CONGEST model) guarantees that all vertices successfully receive m_0. This holds since for every $v \in V$ and a fixed set of adversarial edges F, the family $\mathcal{G}$ contains a subgraph G_i which contains a short s-v path (of length L) and does not contain any of the adversarial edges. Given this observation, our challenge is two folds:

1. provide a round-efficient algorithm for exchanging m_0 over all $\mathcal{G}$-subgraphs simultaneously,
2. guarantee that each vertex outputs the message m_0 while ignoring the remaining messages.

To address the first challenge, we show that the family of subgraphs obtained by this technique has an additional key property of *bounded width*. Informally, a family $\mathcal{G}$ of G-subgraphs has a bounded width if each G-edge appears in all but a bounded number of subgraphs in $\mathcal{G}$. The bounded width of $\mathcal{G}$ allows us to exchange messages in all these subgraphs simultaneously, in a nearly optimal number of rounds. Intuitively, if we propagate messages from the source s over all the subgraphs in a pipeline manner, if we look at an edge (s, v) that participates in a subgraph $G_i \in \mathcal{G}$, the number of messages that might cause a delay in sending the G_i-message that v received from s, is bounded by the number of messages that v received over different edges - i.e., the number of subgraphs in which the edge (s, v) does not participate in. The round complexity of this scheme is based on a very careful analysis which constitutes the key technical contribution in this work. To the best of our knowledge, the bounded width property of the FT sampling technique has been used before only in the context of data structures [193, 81]. It is therefore interesting to see that it finds new applications in the context of reliable distributed communication. The second challenge is addressed by performing an additional communication phase which filters out the corrupted messages. As we will see, the round complexities of our broadcast algorithms for general graphs will be dominated by the cardinality of covering families (which are nearly tight in a wide range of parameters, as shown in [102]).

We also consider the family of expander graphs, which received a lot of attention in the context of distributed resilient computation [61, 187, 106, 12]. For these graphs, we are able to show covering families[1] of considerably smaller cardinality that scales linearly with the number of the adversarial edges. This covering family is obtained by using Karger's edge sampling technique [100], and its conductance-based analysis by Wulff-Nilsen [195]. We hope this result

[1]Using a somewhat more relaxed definition of these families.

will also be useful in the context of FT network design. We next describe our key contribution in more detail.

Our Results

We adopt the gradual approach of fault tolerant graph algorithms, and start by studying broadcast algorithms against a single adversarial edge. Perhaps surprisingly, already this case has been fairly open. We show:

Theorem 12 (Broadcast against a Single Adversarial Edge)**.** *Given a D–diameter, 3 edge-connected graph G, there exists a deterministic algorithm for broadcast against a single adversarial edge that runs in $\widetilde{O}(D^2)$ adversarial-*CONGEST *rounds. In addition, at the end of the algorithm, all vertices also compute a linear estimate for the* diameter *of the graph.*

This improves considerably upon the (implicit) state-of-the-art $n^{O(D)}$ bound obtained by previous algorithms (e.g., by [137, 43]). In addition, in contrast to many previous works (including [137, 43]), our algorithm does not assume global knowledge of the graph or any estimate on the graph's diameter. In fact, at the end of the broadcast algorithm, the vertices also obtain a linear estimate of the graph diameter.

Using the covering family obtained by the standard FT-sampling technique, it is fairly painless to provide a broadcast algorithm with a round complexity of $\widetilde{O}(D^3)$. Our main efforts are devoted to improving the complexity to $\widetilde{O}(D^2)$ rounds. Note that the round complexity of D^3 appears to be a natural barrier for this problem for the following reason. There exists a 3 edge-connected D-diameter graph $G = (V, E)$ and a pair of vertices s, v such that in any collection of 3 edge-disjoint s-v paths P_1, P_2, P_3, the length of the longest path is $\Omega(D^3)$ (Cor. 11, Chapter 6). The improved bound of $\widetilde{O}(D^2)$ rounds is obtained by exploiting another useful property of the covering families of [102]. One can show that, in our context, each G-edge appears on all but $O(\log n)$ many subgraphs in the covering family. This plays a critical role in showing that the simultaneous message exchange on all these subgraphs can be done in $\widetilde{O}(D^2)$ rounds (i.e., linear in the number of subgraphs in this family).

Multiple adversarial edges. We consider the generalization of our algorithms to support f adversarial edges. For $f = O(1)$, we provide broadcast algorithms with $\mathsf{poly}(D)$ rounds.

Theorem 13 (Broadcast against f-Adversarial Edges)**.** *There exists a deterministic broadcast algorithm against f adversarial edges, for every D–diameter $(2f + 1)$ edge-connected graph, with round complexity of $(fD \log n)^{O(f)}$. Moreover, this algorithm can be implemented in $O(fD \log n)$* LOCAL *rounds (which is nearly optimal).*

We note that we did not attempt to optimize for the constants in the exponent in our results for multiple adversarial edges. The round complexity of the algorithm is mainly dominated by the number of subgraphs in the covering family.

Improved broadcast algorithms for expander graphs. We then turn to consider the family of expander graphs, which has been shown to have various applications in the context of resilient distributed computation [61, 187, 106, 12]. Since the diameter of expander graphs is logarithmic, the algorithm of Theorem 13 yields a round complexity of $(f \log n)^{O(f)}$. We provide a considerably improved solution using a combination of tools. The improved broadcast algorithm is designed for ϕ-expander graphs with minimum degree $\Theta(f^2 \log n/\phi)$. One can show that these graphs also have a sufficiently large edge-connectivity.

Theorem 14. *Given an n-vertex ϕ-expander graph with minimum degree $\Theta(f^2 \log n/\phi)$, there exists a randomized broadcast algorithm against $f = O(\sqrt{\frac{n \cdot \phi}{\log n}})$ adversarial edges with round complexity of $O(f \cdot \log^2 n/\phi)$ rounds.*

To obtain this result, we employ the edge sampling technique by Karger [100]. The correctness arguments are based on Wulff-Nilsen [195], who provided an analysis of this technique for expander graphs with large minimum degree. Due to the challenges arose by the adversarial setting, the implementation of the sampling technique requires a somewhat larger minimum degree than that required by the original analysis of [195].

Improved broadcast algorithms with a small shared seed. Finally, we show (nearly) optimal broadcast algorithms given that all the vertices have a shared seed of $\widetilde{O}(1)$ bits.

Theorem 15 (Nearly Optimal Broadcast with Shared Randomness)**.** *There exists a randomized broadcast algorithm against a single adversarial edge that runs in $\widetilde{O}(D)$ rounds, provided that all vertices are given* $\mathsf{poly}(\log n)$ *bits of shared randomness.*

This result is obtained by presenting a derandomization for the well-known fault-tolerant (FT) sampling technique [193]. The FT-sampling technique is quite common in the area of fault-tolerant network design [56], and attracted even more attention recently [152, 38, 31, 102]. While it is relatively easy to show that one can implement the sampling using $\widetilde{O}(D)$ random bits, we show that $\widetilde{O}(1)$ bits are sufficient. This is obtained by using the pseudorandom generator (PRG) of Gopalan [80] and its recent incarnation in distributed settings [153]. We note that for a large number of faults f, the complexity is unlikely to improve from $D^{O(f)}$ to $\widetilde{O}(D)$ even when assuming shared randomness, i.e., the complexity can be improved only by a factor of D. Using the framework of *pseudorandom generator* [146, 188], we provide an improved broadcast algorithm for expander graphs with minimum degree $\Omega(f \cdot \log n/\phi)$ that can tolerate f adversarial edges.

Lemma 11. *Given an n-vertex ϕ-expander graph with minimum degree $\Omega(f \cdot \log n/\phi)$, there exists a randomized broadcast algorithm against f adversarial edges, with a round complexity of $O(f \log^2 n/\phi)$, provided that all vertices have a shared seed of $O(\log n)$ bits.*

Road Map. The broadcast algorithm against a single adversarial edge and the proof of Theorem 12 are given in Section 5.2. In Section 5.3 we consider multiple adversarial edges and prove Theorem 13. In Section 5.4, we show nearly optimal algorithms for expander graphs, providing Theorem 14. Finally, in Section 5.5 we show nearly optimal algorithms assuming shared randomness of $\widetilde{O}(1)$ bits, and prove Theorem 15 and Lemma 11.

5.2 Broadcast Algorithms against a Single Adversarial Edge

In this section, we prove Theorem 12. We first assume, in Section 5.2.1 that the vertices have a linear estimate $c \cdot D$ on the diameter of the graph D, for some constant $c \geq 1$. A-priori, obtaining the diameter estimation seems to be just as hard as the broadcast task itself. In Section 5.2.2, we then show how this assumption can be removed. Throughout, we assume that the message m_0 consists of a single bit. In order to send a $O(\log n)$ bit message, the presented algorithm is repeated for each of these bits (increasing the round complexity by a $O(\log n)$ factor).

5.2.1 Broadcast with a Known Diameter

We first describe the adversarial broadcast algorithm assuming that the vertices share a common[1] linear estimate D' on the diameter D, where $D' \in [D, cD]$ for some constant $c \geq 1$. In Section 5.2.2, we omit this assumption. The underlying objective of our broadcast algorithm is to exchange messages over reliable communication channels that avoid the adversarial edge e'. There are two types of challenges: making sure that all the vertices first *receive* the message m_0, and making sure that all vertices correctly *distinguish* between the true bit and the false one. Alg. $\mathsf{BroadcastKnownDiam}(D')$ has two phases, a *flooding* phase and an *acceptance* phase, which at the high level, handles each of these challenges respectively.

The first phase propagates the messages over an ordered collection of G-subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ where each $G_i \subseteq G$ has several desired properties. Specifically, $\mathcal{G}$ is an (L, k) covering family for $L = O(D') = O(D)$ and $k = 1$ (see Definition 17). An important parameter of $\mathcal{G}$ which determines the complexity of the algorithm is denoted as the *width*.

Definition 20 (Width of Covering Family). *The* width *of a collection of subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$, denoted by $\omega(\mathcal{G})$, is the maximal number of subgraphs avoiding a fixed edge in G. That is,*

$$\omega(\mathcal{G}) = \max_{e \in G} |\{G_i \in \mathcal{G} \mid e \notin G_i\}| .$$

By [102] the covering family of Fact 6 has bounded width.

Observation 7 ([102]). *The width of the (L, k) covering family of Fact 6 has width $\widetilde{O}(L)$.*

Proof. The construction of [102] builds the covering family using a hit-miss hash family $H = \{h : [m] \to [q]\}$ where m is the number of edges, and $|H|, q = O(L \mathsf{poly} \log m)$. Then, for each $h \in H$ and $i \in q$ the subgraph $G_{h,i}$ consists of all edges e' such that $h(e') \neq i$.

[1]That is, all vertices share *the same* diameter estimate D'.

Therefore, an edge e does not appear only in the subgraphs of the form $G_{h,h(e)}$ and there are $|H| = O(L\text{poly}\log m)$ such subgraphs. $\square$

In the following, we present a broadcast algorithm whose time complexity depends on several parameters of the covering family. This will establish the argument assuming that all vertices know a linear bound on the diameter $D' \geq D$.

Theorem 16. *Given is a 3 edge-connected graph G of diameter D, where all vertices know a constant factor upper bound on D, denoted as D'. Assuming that the vertices locally know an $(L, 1)$ covering family $\mathcal{G}$ for $L = 7D'$, there exists a deterministic broadcast algorithm against an adversarial edge with $O(\omega(\mathcal{G}) \cdot L + |\mathcal{G}|)$ rounds.*

Broadcast with a known diameter (Proof of Theorem 16). Given a locally known $(L, 1)$ covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ for $L = 7D'$, the broadcast algorithm has two phases. The first phase consists of $O(L \cdot \omega(\mathcal{G}) + |\mathcal{G}|)$ rounds, and the second phase has $O(L)$ rounds.

Phase 1: Flooding phase. The flooding phase consists of $\ell = |\mathcal{G}|$ sub-algorithms $A_1, \ldots, A_\ell$, where in each algorithm A_i, the vertices propagate messages on the underlying subgraph $G_i \in \mathcal{G}$ that is defined locally by the vertices. The algorithm runs the sub-algorithms $A_1, \ldots, A_\ell$ in a pipeline manner, where in the first round of each sub-algorithm A_i, the source vertex s sends the message (m_0, i) to all its neighbors. For every $i \in \{1, \ldots, \ell\}$, a vertex $u \in V$ that received a message (m', i) from a neighbor w, stores the message (m', i) and sends it to all its neighbors if the following conditions hold: (i) $(w, u) \in G_i$, and (ii) u did not receive a message (m', i) in a prior round[1]. For a vertex u and messages $(m_1, i_1), \ldots, (m_k, i_k)$ waiting to be sent in some round τ, u sends the messages according to the order of the iterations $i_1, \ldots, i_k$ (note that potentially $i_j = i_{j+1}$, and there might be at most two messages with index i_j, namely, $(0, i_j)$ and $(1, i_j)$).

Phase 2: Acceptance phase. The second phase consists of $O(L)$ rounds, in which *accept* messages are sent from the source s to all the vertices in the graph, as follows. In the first round, the source vertex s sends an $\textit{accept}(m_0)$ message to all its neighbors. Then every other vertex $u \in V$ accepts a message m' as its final output and sends an $\textit{accept}(m')$ message to all neighbors, provided that the following conditions hold:

(i) there exists $i \in \{1, \ldots, \ell\}$, such that u stored a message (m', i) in Phase 1;

(ii) u received an $\textit{accept}(m')$ message in Phase 2 from a neighbor w_2, such that $(u, w_2) \notin G_i$.

Since $\mathcal{G}$ is locally known, u can locally verify that $(u, w_2) \notin G_i$. This completes the description of the algorithm.

Correctness. We next prove the correctness of the algorithm. Denote the adversarial edge by $e' = (v_1, v_2)$. We begin with showing that no vertex accepts a wrong message.

[1]If it receives several (m', i) messages in the same round, it will be considered as only one.

Claim 20. *No vertex $u \in V$ accepts a false message $m' \neq m_0$.*

Proof. Assume towards contradiction there exists at least one vertex which accepts a message $m' \neq m_0$ during Phase 2. Let u be the *first* vertex that accepts m'. By first we mean that any other vertex that accepted m', accepted the message in a later round than u, breaking ties arbitrarily. Hence, by Phase 2, u received an $accept(m')$ message from a neighbor w, and stored a message (m', i) in Phase 1, where $(u, w) \notin G_i$. Since u is the first vertex that accepts m', the vertex w did not accept m' in the previous round. We conclude that the edge (w, u) is the adversarial edge, and all other edges are reliable. Because the adversarial edge (w, u) was not included in the i-th graph G_i, all messages of the form (m', i) sent by the adversarial edge in Phase 1 are ignored. Since all other edges are reliable, all vertices ignored the false message (m', i) during the first phase (in case they recieved it), in contradiction to the assumption that u stored (m', i) in Phase 1. □

From Claim 20 we can also deduce that in the case where the adversarial edge initiates a false broadcast, it will not be accepted by any of the vertices.

Corollary 3. *If $e' = (v_1, v_2)$ initiates a broadcast, no message is accepted by any vertex.*

Proof. Since no vertex initiated the broadcast, in the second phase the only vertices that can receive $accept(m)$ messages are v_1 and v_2 over the adversarial edge e'. In addition, since e' also initiates the first phase, for every vertex storing a message (m, i) in Phase 1 it must hold that $e' \in G_i$. Hence, we can conclude that neither v_1 nor v_2 accepts any of the false messages. Consequently, no vertex in $V \setminus \{v_1, v_2\}$ receives an $accepts(m)$ message for any m, as required. □

So far, we have shown that if a vertex v accepts a message, it must be the correct one. It remains to show that each vertex indeed accepts a message during the second phase. Towards that goal, we will show that the collection of ℓ sub-algorithms executed in Phase 1 can be executed in $O(\omega(\mathcal{G}) \cdot L + |\mathcal{G}|)$ rounds. This argument holds regardless of the power of the adversary.

Lemma 12. *Consider an $(L, 1)$ covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ for G that is locally known by all the vertices, and let e' denote the (unknown) adversarial edge. For a fixed vertex v, an edge e, and an L-length s-v path $P \subseteq G' = (V, E \setminus \{e, e'\})$, let $G_i \in \mathcal{G}$ be the subgraph containing P where $e \notin G_i$. Then, v receives the message (m_0, i) in Phase 1 within $O(L \cdot \omega(\mathcal{G}) + |\mathcal{G}|)$ rounds.*

We note that for $D' \geq D$, by Obs.6 taking $L = 7D'$ yields that for every vertex v and edge e, it holds that $\text{dist}_{G \setminus \{e,e'\}}(s, v) \leq L$. Hence, by the properties of the covering family $\mathcal{G}$ (Definition 17), for every vertex v and an edge e there exists an L-length s-v path $P \subseteq G \setminus \{e, e'\}$ and a subgraph G_i that contains P and avoids e. The proof of Lemma 12 is one of the most technical parts in this work. Whereas pipeline is a very common technique, especially in the context of broadcast algorithms, our implementation of it is quite nontrivial. Unfortunately, since our adversary has full knowledge of the randomness of the vertices, it is unclear how to apply the random delay approach of [78, 113] in our setting. We next show that our pipeline approach works well due to the bounded width of the covering family.

Proof of Lemma 12. Let $P = (s = v_0, \ldots, v_\eta = v)$ be an s-v path in $G_i \setminus \{e'\}$, where $\eta \leq L$ and $e \notin G_i$. For simplicity, we consider the case where the only message propagated during the phase is m_0. The general case introduces a factor of 2 in the round complexity. This holds since there could be at most two messages of the form $(0, i)$ and $(1, i)$. We also assume, without loss of generality, that each vertex v_j receives the message (m_0, i) for the *first* time from v_{j-1}. If v_j received (m_0, i) for the first time from a different neighbor in an earlier round, the time it sends the message can only decrease.

In order to show that $v_\eta = v$ receives the message (m_0, i) within $O(L \cdot \omega(\mathcal{G}) + |\mathcal{G}|)$ rounds, it is enough to bound the total number of rounds the message (m_0, i) spent in the *queues* of the vertices of P, waiting to be sent. That is, for every vertex $v_j \in P$, let r_j be the round in which v_j received the message (m_0, i) for the first time, and let s_j be the round in which v_j sent the message (m_0, i). In order to prove Lemma 12, our goal is to bound the quantity

$$T = \sum_{j=1}^{\eta-1} (s_j - r_j). \tag{5.1}$$

For every $k < i$ we denote the set of edges from P that are not included in the subgraph G_k by

$$N_k = \{(v_{j-1}, v_j) \in P \mid (v_{j-1}, v_j) \notin G_k\},$$

and define $\mathcal{N} = \{(k, e) \mid e \in N_k, k \in \{1, \ldots, i-1\}\}$. By the definition of the width property, it holds that:

$$|\mathcal{N}| = \sum_{k=1}^{i-1} |N_k| \leq \eta \cdot \omega(\mathcal{G}) = O(\omega(\mathcal{G}) \cdot L). \tag{5.2}$$

By Eq. (5.1), (5.2) it follows that in order to prove Lemma 12 it is enough to show $T \leq |\mathcal{N}|$. For every vertex $v_j \in P$, let Q_j be the set of messages (m_0, k) that v_j sent between rounds r_j and round s_j. By definition, $|Q_j| = (s_j - r_j)$, and $T = \sum_{j=1}^{\eta-1} |Q_j|$. We next show that $\sum_{j=1}^{\eta-1} |Q_j| \leq |\mathcal{N}|$. This is shown in two steps. First we define a set I_j consisting of certain (m_0, k) messages such that $|Q_j| \leq |I_j|$, for every $j \in \{1, \ldots, \eta - 1\}$. Then, we show that $\sum_{j=1}^{\eta-1} |I_j| \leq |\mathcal{N}|$.

Step one. For every vertex $v_j \in P$, let I_j be the set of messages (m_0, k) satisfying the following three properties : (1) $k < i$, (2) v_j sent the message (m_0, k) before sending the message (m_0, i) in round s_j, and (3) v_j did not receive the message (m_0, k) for the first time from v_{j-1} before round r_j. In other words, the set I_j includes messages received by v_j with a graph index at most $(i-1)$, that are either received from v_{j-1} between round r_j+1 and round s_j-1, or received by v_j from another neighbor $w \neq v_{j-1}$ by round s_j (provided that those messages were not received additionally from v_{j-1}).

Note that it is not necessarily the case that $Q_j \subseteq I_j$. For example, consider a message (m_0, k) where $k < i$, received by v_j from v_{j-1} in some round $r' < r_i$. Hence, by the definition of I_j it holds that $(m_0, k) \notin I_j$. However, it could be the case that v_j sent the message (m_0, k) to v_{j+1} only in some round $s' > r_i$ and $(m_0, k) \in Q_j$. This is because in rounds r' to r_i it was busy sending messages with lower indices. For our purposes, it is sufficient to show the following.

Claim 21. *For every* $1 \leq j \leq \eta - 1$ *it holds that* $|Q_j| \leq |I_j|$.

Proof. We divide the set Q_j into two disjoint sets, $Q_{j,1}$ and $Q_{j,2}$. Let $Q_{j,1}$ be the set of messages (m_0, k) in Q_j that v_j received from v_{j-1} *before* round r_j, and let $Q_{j,2} = Q_j \setminus Q_{j,1}$, i.e., the set of messages in Q_j that v_j did not receive from v_{j-1} by round r_j. We first note that by the definition of the set I_j, it holds that $Q_{j,2} \subseteq I_j$. Let I' be the set of messages (m_0, k) such that (i) $k < i$ (ii) (m_0, k) was not received by v_j from v_{j-1} for the first time, and (iii) v_j sent the message (m_0, k) by round r_i. Hence, it holds that $I' \subseteq I_j \setminus Q_j$. We will next show that $|Q_{j,1}| \leq |I'|$ by describing an injective function f from $Q_{j,1}$ to I'.

For a message (m_0, k) that was received by v_j in round $r_{j,k}$, let $\pi(k)$ be the graph index such that the message $(m_0, \pi(k))$ was sent by v_j in round $r_{j,k}+1$. For every message $(m_0, k) \in Q_{j,1}$, we define $f(m_0, k) = (m_0, \pi^{\alpha_k}(k))$ where α_k is the minimal integer for which $(m_0, \pi^{\alpha_k}(k)) \in I'$. We first show that the function f is well defined. We start with observing that by the definition of $Q_{j,1}$, the message (m_0, k) was received by v_j before round r_j, and was sent only after round r_j. Hence, the message (m_0, k) was not sent by v_j in round $r_{j,k}+1$, and therefore $\pi(k) \neq k$. Moreover, if the message $(m_0, \pi(k))$ was received by v_j for the first time from v_{j-1}, it was received before round $r_{j,k}$ (i.e, $r_{j,\pi(k)} < r_{j,k}$), and therefore $\pi^2(k) \neq \pi(k)$. Using an inductive argument we can conclude that as long as $(m_0, \pi^{\beta}(k))$ was received by v_j from v_{j-1}, it holds that $\pi^{\beta}(k) \neq \pi^{\beta+1}(k)$. Hence, there exists an integer α_k such that $(m_0, \pi^{\alpha_k}(k))$ was not received by v_j from v_{j-1}. Moreover, because the messages are sent according to the order of the indices, it holds that $\pi(\lambda) \leq \lambda$ for every index λ. Therefore $\pi^{\alpha}(k) < i$, and $(m_0, \pi^{\alpha_k}(k)) \in I'$. We conclude that the function f is well defined.

We are left to show that f is an injective. Assume towards contradiction that there exists two different messages $(m_0, k_1), (m_0, k_2) \in Q_{j,i}$ such that $f((m_0, k_1)) = f((m_0, k_2))$. Let β be the smallest integer such that $\pi^{\beta}(k_1) = \pi^{\lambda}(k_2)$, for some integer λ. By the minimality of β it holds that $\pi^{\beta-1}(k_1) \neq \pi^{\lambda-1}(k_2)$. In particular, the messages $(m_0, \pi^{\beta-1}(k_1))$ and $(m_0, \pi^{\lambda-1}(k_1))$ where sent in two different rounds r_1 and r_2. Since $\pi^{\beta}(k_1) = \pi^{\lambda}(k_2)$ it then follows that this message was sent in two different rounds: r_1+1 and r_2+1, in contradiction to the definition of the algortihm. We conclude that f is an injective function and $|Q_{j,1}| \leq |I'|$. The claim follows. □

Step two. We next show that $\sum_{j=1}^{\eta-1} |I_j| \leq |\mathcal{N}|$ by introducing an injection function f from $\mathcal{I} = \{(v_j, k) \mid (m_0, k) \in I_j\}$ to $\mathcal{N}$, defined as follows. For $(v_j, k) \in \mathcal{I}$, we set $f((v_j, k)) = (k, (v_{h-1}, v_h))$ such that (v_{h-1}, v_h) is the closest edge to v_j on $P[v_0, v_j]$, where $(v_{h-1}, v_h) \in N_k$ (i.e., $(v_{h-1}, v_h) \notin G_k$). Specifically,

$$f((v_j, k)) = (k, (v_{h-1}, v_h)) \mid h = \max_{\tau \leq j}\{\tau \mid (v_{\tau-1}, v_\tau) \in N_k\} . \tag{5.3}$$

We begin by showing that the function is well defined.

Claim 22. *The function $f : \mathcal{I} \to \mathcal{N}$ is well defined.*

Proof. For a pair $(v_j, k) \in \mathcal{I}$, we will show that there exists an edge in the path $P[v_0, v_j]$ that is not included in G_k, and therefore $f((v_j, k))$ is defined. Assume by contradiction that all path edges in $P[v_0, v_j]$ are included in G_k. We will show by induction on h that for every $1 \leq h \leq j$ the vertex v_h receives the message (m_0, k) from v_{h-1} before receiving the message (m_0, i) from v_{h-1}, leading to a contradiction with property (3) in the definition of I_j, as $(v_j, k) \in I_j$.

For the base case of $h = 1$, at the beginning of the flooding procedure the source $s = v_0$ sends the messages $(m_0, 1), \ldots, (m_0, i)$ one after the other in rounds $1, \ldots, i$ respectively. Since the edge $(v_0, v_1) \in G_k$, the vertex v_1 receives the message (m_0, k) from v_0 in round k, and in particular before receiving the message (m_0, i) (in round i). Assume the claim holds up to vertex v_h, and we will next show correctness for v_{h+1}. By the induction assumption, v_h receives the message (m_0, k) from v_{h-1} before it receives the message (m_0, i). Hence, v_h also sends (m_0, k) before sending (m_0, i). Because $(v_h, v_{h+1}) \in G_k$, the message (m_0, k) is not ignored, and v_{h+1} receives the message (m_0, k) from v_h before receiving the message (m_0, i). Note that this proof heavily exploits the fact that all the edges on P are reliable. □

Next, we show that the function f is an injection.

Claim 23. *The function f is an injection.*

Proof. First note that by the definition of the function f (see Eq. (5.3)), for every $k_1 \neq k_2$, and $1 \leq j_1, j_2 \leq \eta - 1$ such that $(v_{j_1}, k_1), (v_{j_2}, k_2) \in \mathcal{I}$, it holds that $f((v_{j_1}, k_1)) \neq f((v_{j_2}, k_2))$. Next, we show that for every $k < i$ and $1 \leq j_1 < j_2 \leq \eta - 1$ such that $(v_{j_1}, k), (v_{j_2}, k) \in \mathcal{I}$, it holds that $f((v_{j_1}, k)) \neq f((v_{j_2}, k))$. Denote by $f((v_{j_1}, k)) = (k, (v_{h_1-1}, v_{h_1}))$ and $f((v_{j_2}, k)) = (k, (v_{h_2-1}, v_{h_2}))$. We will now show that $(v_{h_2-1}, v_{h_2}) \in P[v_{j_1}, v_{j_2}]$. Since $(v_{h_1-1}, v_{h_1}) \in P[v_0, v_{j_1}]$, it will then follow that $(v_{h_1-1}, v_{h_1}) \neq (v_{h_2-1}, v_{h_2})$.

Assume towards contradiction that $(v_{h_2-1}, v_{h_2}) \in P[v_0, v_{j_1}]$. By the definition of f and the maximality of h_2, it holds that $P[v_{j_1}, v_{j_2}] \subseteq G_k$. Since $(v_{j_1}, k) \in \mathcal{I}$, the vertex v_{j_1} sent the message (m_0, k) before sending (m_0, i). Since we assumed every vertex $v_t \in P$ receives the message (m_0, i) for the first time from v_{t-1} (its incoming neighbor on the path P), and all the edges on P are reliable, it follows that v_{j_2} received the message (m_0, k) from v_{j_2-1} *before* receiving the message (m_0, i) in round r_{j_2}. This contradicts the assumption that $(v_{j_2}, k) \in \mathcal{I}$, as by property (3) in the definition of I_{j_2}, the vertex v_{j_2} did not receive the message (m_0, k) from v_{j_2-1} before round r_{j_2}. □

This completes the proof of Lemma 12. Finally, we show that when $D' \geq D$, all vertices accept the message m_0 during the second phase using Lemma 12. This will conclude the proof of Theorem 16.

Claim 24. *All vertices accept m_0 within $O(L)$ rounds from the beginning of the second phase, provided that $D' \geq D$.*

Proof. Let T be a BFS tree rooted at s in $G \setminus \{e'\}$. For a vertex u, let $p(u)$ be the parent of u in the tree T. We begin with showing that each vertex u receives and stores a message (m_0, j) such that $(u, p(u)) \notin G_j$ during the first phase. Because the graph is 3 edge-connected, by Obs. 6 for every vertex u there exists an s-v path P in G that does not contain both the edge $(u, p(u))$ and e', of length $|P| \leq 7D \leq 7D' = L$. By the definition of the covering family $\mathcal{G}$, there exists a subgraph G_j containing all the edges in P, and in addition, $(u, p(u)) \notin G_j$. Hence, by Lemma 12, u stores a message of the form (m_0, j) during the first phase.

We next show by induction on i that all vertices in layer i of the tree T accepts m_0 by round i of Phase 2. For the base case of $i = 1$, given a vertex u in the first layer of T, u receives the message $accept(m_0)$ from s after one round of Phase 2. Since u stored a message (m_0, j) such that $(s, u) \notin G_j$ in Phase 1, it accepts the message m_0 within one round. Assume the claim holds for all vertices up to layer $(i - 1)$ and let u be a vertex in the i-th layer. By the induction assumption, $p(u)$ sent $accept(m_0)$ to u by round $(i - 1)$. Because u stored a message (m_0, j) such that $(p(u), u) \notin G_j$ during the first phase, u accepts m_0 by round i. □

Remark. Our broadcast algorithm does not need to assume that the vertices know the identity of the source vertex s. By Cor. 3, in case where the adversarial edge e' initiates a false broadcast execution, no vertex will accept any of the messages sent.

5.2.2 Broadcast without Knowing the Diameter

We next show how to remove the assumption that the vertices know an estimate of the diameter; consequently, our broadcast algorithm also computes a linear estimate of the diameter of the graph. This increases the round complexity by a logarithmic factor and establishes Theorem 12. We first describe the algorithm under the simultaneous wake-up assumption and then explain how to remove it.

Alg. Broadcast**.** The algorithm applies Alg. BroadcastKnownDiam of Section 5.2 for $k = O(\log D)$ iterations in the following manner. Every iteration $i \in \{1, \ldots, k\}$ consists of the following steps.

Step 1: In the first step, the source vertex s initiates Alg. $\mathsf{BroadcastKnownDiam}(D_i)$ with diameter estimate $D_i = 2^i$, and the desired message m_0. This step consists of $r_{i,1} = \widetilde{O}(D_i^2)$ rounds, where $r_{i,1}$ be an upper bound on the round complexity of $\mathsf{BroadcastKnownDiam}(D_i)$.

Denote all vertices that accepted the message m_0 by the end of the first step by A_i and let $N_i = V \setminus A_i$ be the set of vertices that did not accept the message by the end of the step. We note that since this step consists of a fixed number of $r_{i,1}$ rounds, by the end of the first step every vertex can deduce whether it is in A_i or N_i.

Step 2: In the second step, the vertices in N_i inform s that the computation is not yet completed in the following manner. At the beginning of the step, every vertex in N_i broadcast *the same* designated message M_C by applying Alg. $\mathsf{BroadcastKnownDiam}(9D_i)$ with diameter estimate $9D_i$ and the message M_C. This step consists of $r_{i,2} = \widetilde{O}(D_i^2)$ rounds, where $r_{i,2}$ be an upper bound on the round complexity of Alg. $\mathsf{BroadcastKnownDiam}(9D_i)$.

Step 3: In the third step, the vertices decide whether to continue to the next iteration or halt. This step consists of $r_{i,3}$ rounds where $r_{i,3}$ is an upper bound on the round complexity of Alg. $\mathsf{BroadcastKnownDiam}(28D_i)$. In case the source vertex s did not receive and accept the designated message M_C during the second step, it broadcasts a termination message M_T to all vertices in V by applying Alg. $\mathsf{BroadcastKnownDiam}(28D_i)$ with diameter estimate $28D_i$. In case the source vertex s accepted the message M_C during the second step, at the end of this step (i.e, after $r_{i,3}$ many rounds), s continues to the next iteration $(i + 1)$.

A vertex $v \in V$ that accepts the termination message M_T during this step completes the execution with the output message it has accepted so far. Additionally, v considers D_i as an estimation of the graph diameter. If by the end of the step v did not accept the termination message M_T, it continues to iteration $(i + 1)$.

Analysis. We begin with noting that no vertex $v \in V$ accepts a wrong message $m' \neq m_0$ as its output. This follows by Claim 20 and the correctness of Alg. $\mathsf{BroadcastKnownDiam}$.

Observation 8. *No vertex $v \in V$ accepts a wrong message $m' \neq m_0$.*

Fix an iteration i. Our next goal is to show that if $N_i \neq \emptyset$, then s will accept the message M_C by the end of Step 2. Consider the second step of the algorithm, where the vertices in N_i broadcast the message M_C towards s using Alg. $\mathsf{BroadcastKnownDiam}(9D_i)$. Since all vertices in N_i broadcast *the same* message M_C, we refer to the second step as a *single* execution of Alg. $\mathsf{BroadcastKnownDiam}(9D_i)$ with multiple sources. We begin with showing that the distance between the vertices in A_i and s is at most $14D_i$.

Claim 25. *For every vertex $v \in A_i$ it holds that* $\text{dist}_{G\setminus\{e'\}}(s, v) \leq 14D_i$.

Proof. Recall that Alg. $\mathsf{BroadcastKnownDiam}$ proceeds in two phases. In the first phase, the source vertex propagates messages of the form (m, k), and in the second phase, the source vertex propagates *accept* messages. For a vertex $v \in A_i$ that accepts the message m_0 in the first step of the i-th iteration, by the second phase of Alg. $\mathsf{BroadcastKnownDiam}(D_i)$, it received an $accept(m_0)$ message from a neighbor w in Phase 2, and stored a message (m_0, k) in Phase 1, such that $(v, w) \notin G_k$. Let P_1 be the path on which the message $accept(m_0)$ propagated towards v in Phase 2 of Alg. $\mathsf{BroadcastKnownDiam}(D_i)$. Since the second phase is executed for $7D_i$ rounds, it holds that $|P_1| \leq 7D_i$. In the case where $e' \notin P_1$, since s is the only vertex initiating $accept(m_0)$ messages (except maybe e'), P_1 is a path from s to v in $G \setminus \{e'\}$ as required.

Assume that $e' \in P_1$, and denote it by $e' = (v_1, v_2)$. Without loss of generality, assume that on the path P_1, the vertex v_1 is closer to v than v_2. Hence, v_1 received an $accept(m_0)$ message from v_2 during Phase 2, and because v_1 also sent the message over P_1, it accepted m_0 as its output. Therefore, during the execution of Alg. $\mathsf{BroadcastKnownDiam}(D_i)$, the vertex v_1 stored a message (m_0, j) during the first phase, where $e' \notin G_j$. As all edges in G_j are reliable, we conclude that G_j contains an s-v_1 path P_2 of length $\eta \leq 7D_i$ such that $e' \notin P_2$. Thus, the concatenated path $P_2 \circ P_1[v_1, v]$ is a path of length at most $14D_i$ from s to v in $G \setminus \{e'\}$ as required. $\square$

We now show that if $N_i \neq \emptyset$ then s accepts the message M_C during the second step and continues to the next iteration. The proof is very similar to the proof of Claim 24 and follows from the following observation.

Observation 9. *For every $u \in A_i$ and edge $e = (v, u)$, it holds that* $\text{dist}_{G \setminus \{e', e\}}(N_i, u) \leq 7 \cdot 9D_i$.

Proof. Let T be a truncated BFS tree rooted at s in $G \setminus \{e'\}$, such that (1) $A_i \subseteq V(T)$, and (2) the leaf vertices of T are in the A_i set. Informally, T is a minimum depth tree rooted at s in $G \setminus \{e'\}$, that spans the vertices in A_i. By Claim 25 the depth of T is at most $14D_i$. It follows that the forest $T \setminus \{e\}$ contains at most two trees of diameter $2 \cdot 14D_i$. Since G is 3 edge-connected, there exists a path from some vertex in N_i to u in $G \setminus \{e, e'\}$.

Hence, the shortest path from N_i to u in $G \setminus \{e, e'\}$ denoted as P can be transformed into a path P' containing at most one edge in P, and two tree subpaths of the forest $T \setminus \{e\}$. Therefore, $\text{dist}_{G \setminus \{e, e'\}}(N_i, u) \leq |P'| \leq 4 \cdot 14D_i + 1 \leq 7 \cdot 9D_i$. $\square$

Claim 26. *If $N_i \neq \emptyset$, s accepts the message M_C by the end of Step 2 of the i-th iteration.*

Proof. Let $P = (u_0, u_1, \ldots, u_\eta = s)$ be a shortest path from some vertex $u_0 \in N_i$ to the source vertex s in $G \setminus \{e'\}$. As P is the shortest such path, for every $j \neq 0$ $u_j \in A_i$, and by Claim 25 $|P| \leq (14D_i + 1)$. In order to prove Claim 26, we will show by induction on j that every $u_j \in P$ accepts the message M_C by round j of Phase 2 in Alg. $\mathsf{BroadcastKnownDiam}(9D_i)$ (in Step 2).

For the base case of $j = 0$, as $u_0 \in N_i$, it accepts the message M_C at the beginning of the phase. Assume the claim holds for u_j and consider the vertex u_{j+1}. By Obs. 9 there exists a path P_{j+1} from some vertex in N_i to u_{j+1} in $G \setminus \{e', (u_j, u_{j+1})\}$ of length $|P_{j+1}| \leq 7 \cdot 9D_i$. Hence, by Lemma 12 combined with the covering family used in Alg. $\mathsf{BroadcastKnownDiam}(9D_i)$, we conclude that u_{j+1} stored a message (M_C, k) in Phase 1 where $(u_j, u_{j+1}) \notin G_k$. By the induction assumption, u_j sends u_{j+1} an $accept(M_C)$ message by round j of Phase 2. Since $(u_{j+1}, u_j) \notin G_k$, it follow that u_{j+1} accepts the message M_C by round $(j+1)$. □

By Claim 25 it follows that when $D_i < (D/28)$ there must exist a vertex $w \in V$ that did not accept the message m_0 during the execution of Alg. $\mathsf{BroadcastKnownDiam}(D_i)$ in the first step, and $N_i \neq \emptyset$. On the other hand, when $D_i \geq D$ by Claim 24 all vertices accept the message m_0 during the first step of the i-th iteration, and therefore $N_i = \emptyset$. Hence, for an iteration i^* in which no vertex broadcasts the message M_C (and therefore s decides to terminate the execution in Step 3), it holds that $D_{i^*} \in [D/28, 2D]$. Since s broadcasts the termination message M_T in Step 3 by applying Alg. $\mathsf{BroadcastKnownDiam}(28D_{i^*})$ with diameter estimate $28D_{i^*}$, we conclude that all vertices in V will finish the execution as required.

Omitting the simultaneous wake-up assumption. The main adaptation is that in the second step of each iteration, the message M_C is initiated by the vertices in N_i with neighbors in A_i (and possibly also the endpoints of the adversarial edge), rather than all the vertices in N_i. Specifically, the modifications are as follows. At the end of the first step, every vertex $u \in A_i$ informs all its neighbors that the first step has ended. Every vertex $u \in N_i$ receiving such a message, initiates the second step by broadcasting the message M_C using Alg. $\mathsf{BroadcastKnownDiam}(9D_i)$. In the case where some vertex $u \in N_i$ receives messages indicating that the first step has ended from two distinct neighbors w, w' in two distinct rounds $\tau \neq \tau'$, the vertex u broadcasts the message M_C in rounds $(\tau + 1)$ and $(\tau' + 1)$. We note that this case can occur when at least one of the endpoints of the adversarial edge e' is in N_i. A vertex $u \in A_i$ that receives M_C during the first step ignores these messages.

The correctness follows by the fact that for every edge $(u, w) \in G \setminus \{e'\}$ such that $u \in N_i$ and $w \in A_i$, the vertex u broadcasts the message M_C at the beginning of the second step of iteration i. For that reason, Claim 26 works in the same manner. Additionally, in an iteration i^* where $N_{i^*} = \emptyset$, no vertex broadcasts the message M_C in the second step, and therefore s broadcasts the termination message M_T.

5.3 Broadcast against f Adversarial Edges

In this section, we consider the broadcast problem against f adversarial edges and prove Theorem 13. The adversarial edges are fixed throughout the execution but are unknown to any of the vertices. Given a D–diameter, $(2f + 1)$ edge-connected graph G, and at most f adversarial edges $F \subseteq E$, the goal is for a source vertex s to deliver a message m_0 to all vertices in the graph. At the end of the algorithm, each vertex is required to output the message m_0. Our algorithm is again based on a locally known covering family $\mathcal{G}$ with several desired properties. The algorithm floods the messages over the subgraphs of $\mathcal{G}$. The messages exchanged over each subgraph $G_i \in \mathcal{G}$ also contains the path information along which the message has been received. As we will see, the round complexity of the algorithm is mostly dominated by the cardinality of $\mathcal{G}$. Towards proving Theorem 13, we prove the following theorem which will become useful also for the improved algorithms for expander graphs in Section 5.4.

Theorem 17. *Given is a $(2f+1)$ edge-connected graph G of diameter D, and a parameter L satisfying that for every $u, v \in V$, and every set $E' \subseteq E$ of size $|E'| \leq 2f$, it holds that $\text{dist}_{G\setminus E'}(u,v) \leq L$. Assuming that the vertices locally know an $(L, 2f)$ covering family $\mathcal{G}$, there exists a deterministic broadcast algorithm* $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}, L, f)$ *against f adversarial edges F with round complexity $O(L \cdot |\mathcal{G}|)$.*

We note that by Obs. 6, every $(2f+1)$ edge-connected graph G with diameter D satisfies the promise of Theorem 17 for $L = (6f+2)D$. We are now ready to describe the broadcast algorithm given that the vertices know an $(L, 2f)$ covering family $\mathcal{G}$ (along with the parameters L and f) as specified by Theorem 17. Later, we explain the general algorithm that omits this assumption.

Alg. $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}, L, f)$**.** Similarly to the single adversarial edge case, the algorithm has two phases, a flooding phase, and an acceptance phase. In the first phase of the algorithm, the vertices exchange messages over the subgraphs of $\mathcal{G}$, which also contains the path information along which the messages are received. In addition, instead of propagating the messages of distinct G_i subgraphs in a pipeline manner, we run the entire i-th algorithm (over the edges of the graph G_i) after finishing the application of the $(i-1)$-th algorithm[1].
In the first phase, the vertices flood *heard bundles* over all the $G_i \in \mathcal{G}$ subgraphs, defined as follows.
Heard bundles: A *bundle* of heard messages sent from a vertex v to u consists of:

1. A header message $heard(m, len, P)$, where P is an s-v path of length len along which v received the message m.

2. A sequence of len messages specifying the edges of P, one by one.

This bundle contains $(len + 1)$ messages that will be sent in a pipeline manner in the following way. The first message is the header $heard(m, len, P)$. Then in the next consecutive len rounds, v sends the edges of P in reverse order (from the edge incident to v to s).
Phase 1: Flooding. The first phase consists of $\ell = |\mathcal{G}|$ iterations, where each iteration is implemented using $O(L)$ rounds. In the first round of the i-th iteration, the source vertex s sends the message $heard(m_0, 1, \emptyset)$ to all neighbors. Every vertex v, upon receiving the *first* bundle message $heard(m', x, P)$ over an edge in G_i from a neighbor w, stores the bundle $heard(m', x+1, P \cup \{w\})$ and sends it to all neighbors. Note that each vertex stores and sends at most *one* heard bundle $heard(m', x, P)$ in each iteration. That is, each vertex stores at most one message per subgraph G_i.

[1]One might optimize the $O(f)$ exponent by employing a pipeline approach in this case as well.

Phase 2: Acceptance. The second phase consists of $O(L)$ rounds, in which *accept* messages are propagated from the source s to all vertices as follows. In the first round, s sends $\mathit{accept}(m_0)$ to all neighbors. Every vertex $v \in V \setminus \{s\}$ decides to accept a message m' if the following two conditions hold: (i) v receives $\mathit{accept}(m')$ from a neighbor w, and (ii) $\text{MinCut}(s, v, \mathcal{P}) \geq f$, where

$$\mathcal{P} = \{P \mid v \text{ stored a } \mathit{heard}(m', \mathit{len}, P) \text{ message and } (v, w) \notin P\} . \tag{5.4}$$

Note that since the decision here is made by computing the minimum cut of a path collection, it is indeed required (by this algorithm) to send the path information.

Correctness. We begin with showing that no vertex accepts a false message.

Claim 27. *No vertex $v \in V$ accepts a message $m' \neq m_0$ in the second phase.*

Proof. Assume by contradiction there exists a vertex that accepts a false message m', and let v be the *first* such vertex. By first we mean that any other vertex that accepted m' accepted the message in a later round than v, breaking ties arbitrarily. Hence, v received a message $\mathit{accept}(m')$ from some neighbor w. Because v is the first such vertex, the edge (w, v) is adversarial. Let $E' = F \setminus \{(w, v)\}$ be the set of the remaining $(f - 1)$ adversarial edges, and let $\mathcal{P}$ be given as in Eq. (5.4). We next claim that $\text{MinCut}(s, v, \mathcal{P}) \leq (f - 1)$ and therefore v does not accept m'.

To see this, observe that any path P such that v received a message $\mathit{heard}(m', \mathit{len}, P)$ must contain at least one edge in E'. This holds even if the content of the path P is corrupted by the adversarial edges. Since there are at most $(f - 1)$ edges in E', and each of the paths in $\mathcal{P}$ intersects these edges, it holds that $\text{MinCut}(s, v, \mathcal{P}) \leq (f - 1)$ as required. □

Finally, we show that all vertices in V accept the message m_0 during the second phase. This completes the proof of Theorem 17.

Claim 28. *All vertices accept m_0 within $O(L)$ rounds from the beginning of the second phase.*

Proof. We will show that all vertices accept the message m_0 by induction on the distance from the source s in the graph $G \setminus F$. The base case holds vacuously, as s accepts the message m_0 in round 0. Assume all vertices at distance at most i from s in $G \setminus F$ accepts the message m_0 by round i. Consider a vertex v at distance $(i + 1)$ from s in $G \setminus F$. By the induction assumption on i, v receives the message $\mathit{accept}(m_0)$ from a neighbor w in round $j \leq (i + 1)$ over a reliable edge (w, v). We are left to show that $\text{MinCut}(s, v, \mathcal{P}) \geq f$, where $\mathcal{P}$ is as given by Eq. (5.4). Alternatively, we show that for every edge set $E' \subseteq E \setminus \{(w, v)\}$ of size $(f - 1)$, the vertex v stores a heard bundle containing m_0 and a path P_k such that $P_k \cap (E' \cup \{(v, w)\}) = \emptyset$ during the first phase. This necessary implies that the minimum cut is at least f.

For a subset $E' \subseteq E$ of size $(f - 1)$, as $|F \cup E' \cup \{w, v\}| \leq 2f$ by the promise on L in Theorem 17, $\text{dist}_{G \setminus (F \cup E' \cup \{w,v\})}(s, v) \leq L$. By the properties of the covering family $\mathcal{G}$ (Definition 17), it follows that there exists a subgraph G_k such that $G_k \cap (F \cup E' \cup \{(v, w)\}) = \emptyset$, and $\text{dist}_{G_k}(s,v) \leq L$. Hence, all edges in G_k are reliable, and the only message that passed through the heard bundles during the k-th iteration is the correct message m_0. As $\text{dist}_{G_k}(s,v) \leq L$, the vertex v stores a heard bundle $heard(m_0, x, P_k)$ during the k-th iteration for some s-v path P_k of length $x = O(L)$. Moreover, as $P_k \subseteq G_k$ it also holds that $P_k \cap (E' \cup \{(v,w)\}) = \emptyset$. We conclude that $\text{MinCut}(s, v, \mathcal{P}) \geq f$, and by the definition of Phase 2, v accepts m_0 by round $(i+1)$. The claim follows as the diameter of $G \setminus F$ is $O(L)$ due to the promise on L. □

Alg. Broadcast **(Proof of Theorem 13)** We now describe the general broadcast algorithm. Our goal is to apply Alg. BroadcastKnownCovFamily$(\mathcal{G}, L, f)$ over the $(L, 2f)$ covering family $\mathcal{G}$ for $L = O(fD)$, constructed using Fact 6. Since the vertices do not know the diameter D (or a linear estimate of it), we make $O(\log D)$ applications of Alg. BroadcastKnownCovFamily$(\mathcal{G}_i, L_i, f)$ using the $(L_i, 2f)$ covering family $\mathcal{G}_i$ for $L_i = O(fD_i)$, where $D_i = 2^i$ is the diameter guess for the i-th application.

Specifically, at the beginning of the i-th application, the source s initiates the execution of Alg. BroadcastKnownCovFamily$(\mathcal{G}_i, L_i, f)$ with the desired message m_0 over an $(L_i, 2f)$ covering family $\mathcal{G}_i$ constructed using Fact 6 with $L_i = O(fD_i)$ and $D_i = 2^i$. Denote all vertices that accepted the message m_0 at the end of Alg. BroadcastKnownCovFamily$(\mathcal{G}_i, L_i, f)$ by A_i, and let $N_i = V \setminus A_i$ be the vertices that did not accept the message.

The algorithm now applies an additional step where the vertices in N_i inform s that they did not accept any message in the following manner. All vertices in N_i broadcast *the same* designated message M by applying Alg. BroadcastKnownCovFamily$(\mathcal{G}'_i, ctL_i, f)$ over a $(ctL_i, 2f)$ covering family $\mathcal{G}'_i$, for some fixed constant $c > 0$ (known to all vertices). This can be viewed as performing a single broadcast execution (i.e., with the same source message) but from $|N_i|$ multiple sources. We next set $\tau_i = O(f \cdot D_i \log n)^{O(f)}$ as a bound on the waiting time for a vertex to receive any acknowledgment.

If the source vertex s accepts the message M at the end of this broadcast execution, it waits for τ_i rounds, and then continues to the next application[1] $(i+1)$ (with diameter guess 2^{i+1}). In the case where s did not accept the message M within τ_i rounds from the beginning of that broadcast execution, it broadcasts a termination message M_T to all vertices in V. This is done by applying Alg. BroadcastKnownCovFamily$(\mathcal{G}'_i, ctL_i, f)$ over the $(ctL_i, 2f)$ covering family $\mathcal{G}'_i$. Once a vertex $v \in V$ accepts the termination message M_T, it completes the execution with the last message it has accepted so far (in the analysis part, we show that it indeed accepts the right message). A vertex v that did not receive a termination message M_T within τ_i rounds, continues to the next application of Alg. BroadcastKnownCovFamily.

The correctness argument exploits the fact that for an application i such that $N_i \neq \emptyset$, the

[1]We make the source vertex s wait since in the case where it actually sends a termination message, all vertices accept it within τ_i rounds. Therefore, we need to make sure that all vertices start the next $(i+1)$ application at the same time.

graph G' obtained by contracting [1] all vertices in N_i into a single vertex a, satisfies the following: (i) it is $(2f+1)$ edge-connected, (ii) it contains s, and (iii) it has diameter $O(L_i) = O(f \cdot D_i)$. Property (i) holds since G is $(2f+1)$ edge-connected, and property (ii) holds by the definition of A_i. We next show property (iii) holds as well.

Claim 29. *For every application i of Alg.* $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}_i, L_i, f)$ *and every vertex $v \in A_i$, it holds that* $\text{dist}_{G \setminus F}(s, v) = O(L_i)$.

Proof. Recall that Alg. $\mathsf{BroadcastKnownCovFamily}$ proceeds in two phases. In the first phase, the source vertex propagates *heard* bundles, and in the second phase, the source vertex propagates *accept* messages. Let v be a vertex that accepts the message m_0 in the i-th application of Alg. $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}_i, L_i, f)$. By Phase 2 of Alg. $\mathsf{BroadcastKnownCovFamily}$, v received an $accept(m_0)$ message from a neighbor w in the second phase, and in the first phase it received heard bundles regarding m_0 over a path collection $\mathcal{P}$ such that $\text{MinCut}(s, v, \mathcal{P}) \geq t$, and $(v, w) \notin P$ for every $P \in \mathcal{P}$. Let P_1 be the path on which the message $accept(m_0)$ propagated towards v in the second phase of Alg. $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}_i, L_i, f)$, executed by s with the desired message m_0. Since the second phase is executed for $O(L_i)$ rounds, it holds that $|P_1| = O(L_i)$. If $P_1 \cap F = \emptyset$, P_1 starts at s and the claim follows.

Next, assume $P_1 \cap F \neq \emptyset$. Let (v_1, v_2) be the closest edge to v on P_1 such that $(v_1, v_2) \in F$ (i.e., $P_1[v_2, v] \cap F = \emptyset$). Hence, v_2 sent an $accept(m_0)$ message to v during the second phase over the path P_1. Therefore, v_2 received heard bundles regarding m_0 over a path collection $\mathcal{P}$ such that $\text{MinCut}(s, v, \mathcal{P}) \geq f$, and $(v_1, v_2) \notin P$ for every $P \in \mathcal{P}$. As $|F \setminus \{(v_1, v_2)\}| \leq (f-1)$, there must be a path $P_2 \in \mathcal{P}$ such that $P_2 \cap F = \emptyset$. Hence, P_2 is an s-v_2 path of length $|P_2| = O(L_i)$ in $G \setminus F$. Thus, the concatenated path $P_2 \circ P_1[v_2, v]$ is a path of length $O(L_i)$ from s to v in $G \setminus F$. $\square$

Using similar arguments to Obs. 6, from Claim 29 it follows that for every set $E' \subseteq E$ of size $|E'| \leq t$, and every vertex $v \in A_i$ it holds that $\text{dist}_{G \setminus F \cup E'}(N_i, v) = O(f \cdot L_i)$.

Observation 10. *For every vertex $v \in A_i$ and an edge set* E' $\subseteq E$ *of size $|E'| \leq f$, it holds that* $\text{dist}_{G \setminus (F \cup E')}(N_i, v) = O(f \cdot L_i)$.

Proof. Let T be a truncated BFS tree rooted at s in $G \setminus F$, such that (1) $A_i \subseteq V(T)$, and (2) all the leaf vertices of T are included in A_i. By Claim 29 the depth of T is $O(L_i)$. It then follows that the forest $T \setminus E'$ consists of $|E'| + 1 \leq (f+1)$ trees, each of diameter $O(L_i)$. Since G is $(2f+1)$ edge-connected, there exists a path from some vertex in N_i to v in $G \setminus (F \cup E')$. Hence, the shortest path from N_i to v in $G \setminus (F \cup E')$ denoted as P can be transformed into a path P' containing at most f edges in P, as well as, $(f+1)$ tree subpaths of the forest $T \setminus E'$. Therefore, $\text{dist}_{G \setminus (F \cup E')}(N_i, v) = O(f \cdot L_i)$. $\square$

[1] I.e., we contract all edges with both endpoints in N_i.

We are now ready to complete the proof of Theorem 13.

Proof of Theorem 13. Fix an application i of Alg. $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}_i, L_i, f)$. We first claim that if $N_i \neq \emptyset$, the source vertex s accepts the message M, and continues to the next application $(i + 1)$. In the second step of the i-th application, the N_i vertices broadcast the message M by executing Alg. $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}'_i, cfL_i, f)$ over a $(cfL_i, 2f)$ covering family $\mathcal{G}'_i$. By Obs. 10, for a large enough constant c it hold that for every $v \in A_i$ and $\mathrm{E}' \subseteq E$ of size $|E'| \leq f$, it holds that $\mathrm{dist}_{G \setminus F \cup E'}(N_i, v) \leq ctL_i$. Hence, using the same arguments as in Claim 28 we can conclude that all vertices in A_i accept the message M in the second phase of Alg. $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}'_i, cfL_i, f)$. Specifically, the source vertex $s \in A_i$ accepts the message and proceeds to application $(i + 1)$.

Next, consider the first application i^* for which $N_{i^*} = \emptyset$, and all vertices accept the broadcast message m_0. Since the diameter of the graph is D, the algorithm must have run at least $D/2$ rounds. This implies that $D_{i^*} = \Omega(D/f)$ and $L_{i^*} = \Omega(D)$. Hence, for a large enough constant c, when s broadcasts the termination message M_T using Alg. $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}'_{i^*}, cfL_{i^*}, f)$ over the $(cfL_{i^*}, 2f)$ covering family $\mathcal{G}'_{i^*}$, all vertices accept M_T and finish the execution.

We next consider the round complexity. Clearly, when taking $D_i \geq D$, by Theorem 17 the set $N_i = \emptyset$, and we are done. Therefore, we can conclude that $i^* \leq \log D + 1$. The algorithm performs $O(\log D)$ applications of Alg. $\mathsf{BroadcastKnownCovFamily}$, each with parameters $L_i = O(fD)$ and $cfL_i = O(f^2 D)$ (used in the construction of $\mathcal{G}'_i$). Thus, the total round complexity is bounded by $(fD \log n)^{O(f)}$. □

Finally, we observe that our broadcast algorithm can be implemented in the LOCAL model using $O(fD \log n)$ many rounds.

Corollary 4. *For every $(2f + 1)$ edge-connected graph, and a source vertex s, there is a deterministic broadcast algorithm against f adversarial edges that runs in $O(fD \log n)$ local rounds.*

Proof. The algorithm is the same as in the $\mathsf{CONGEST}$ model. However, since in the local model there are no bandwidth restrictions, the message propagation over the $|\mathcal{G}|$ subgraphs of the (L, f) covering family can be implemented simultaneously within $L = O(fD)$ rounds. □

5.4 Broadcast Algorithms for Expander Graphs

In this section, we show improved constructions of covering families for expander graphs, which consequently lead to considerably faster broadcast algorithms. Specifically, we show that the family of expander graphs with a sufficiently large minimum degree admits covering families of a considerably smaller size (than that obtained for general graphs). We start by providing the precise definition of expander graphs used in this section.

Expander graphs. For a vertex subset $S \subseteq V$, denote by $\delta_G(S) = |E \cap (S \times (V \setminus S))|$ the number of edges crossing the $(S, V \setminus S)$ cut . The *volume* $vol_G(S)$ of S in G is the number of edges of G incident to S. Assuming both S and $V \setminus S$ have positive volume in G, the *conductance* $\phi_G(S)$ of S is defined as $\phi_G(S) = \delta_G(S)/\min\{vol_G(S), vol_G(V \setminus S)\}$. In case $vol_G(S) = 0$ or $vol_G(V \setminus S) = 0$, the conductance of S is set to be $\phi_G(S) = 0$. The *edge expansion* of a graph G is given by $\phi(G) = \min_{S \subset V} \phi_G(S)$. We say a graph G is a ϕ-expander if $\phi(G) \geq \phi$.

The structure of this section is as follows. We first provide a combinatorial randomized construction of $(L, 2f)$ covering families for n-vertex ϕ-expander graphs[1] with minimum degree $\Theta(f \log n/\phi)$. Then, we show a randomized construction of $(L, 2f)$ covering families in the adversarial CONGEST model. Finally, we provide an improved broadcast algorithm that uses these families and is resilient against f adversarial edges, given a ϕ-expander graph with minimum degree $\Theta(f^2 \log n/\phi)$.

Covering families with improved bounds. The computation of the improved covering family is based on showing that a sampled subgraph of G obtained by sampling each edge independently with probability $p = \Theta(1/f)$ satisfies some desired expansion and connectivity properties.

We use the following result from [195] which provides conductance guarantees for the subgraphs obtained by Karger's edge sampling technique. This result also implies that expander graphs with large minimum degree have large edge-connectivity.

Theorem 18 (Lemma 20 from [195]). *Given $c > 0, \kappa \geq 1$ and $\rho \leq 1$, let $G = (V, E)$ be an n-vertex graph with degree at least $\kappa\rho$. Let $G' = (V, E')$ be the multigraph obtained from G by sampling each edge independently with probability*

$$p = \min\{1, (12c + 24) \ln n/(\rho^2 \kappa)\} .$$

Then, with probability $1 - O(1/n^c)$, for every cut $(S, V \setminus S)$ in G, it holds that:

- *if $\phi_G(S) \geq \rho$ then $\phi_{G'}(S)$ deviates from $\phi_G(S)$ by a factor of at most 4, and*
- *if $\phi_G(S) < \rho$ then $\phi_{G'}(S) < 6\rho$.*

We will use the following fact that appeared in [181] that provides an upper bound for the diameter of ϕ-expander graphs. For completeness we provide here the proof.

Fact 7 (Section 19.1 from [181]). *The diameter of an n-vertex ϕ-expander graph G is bounded by $O(\log n/\phi)$.*

[1]with slightly weaker properties, which are sufficient for our purposes.

Proof. For an integer $k \geq 0$ and a vertex w, let $B_k(w) = \{v \in V \mid \text{dist}_G(w,v) \leq k\}$ be the set of vertices at distance at most k from w. We show by induction on $k \geq 0$ that for every vertex w satisfying that $vol_G(B_k(w)) \leq |E|/2$, it holds that $vol_G(B_k(w)) \geq (1+\phi)^k$. For the base case of $k = 0$, $B_0(w) = \{w\}$. Since $\phi > 0$, the vertex w has at least one neighbor in G, and thus $vol_G(B_0(w)) \geq 1$. Assume that the claim holds for k and consider $B_{k+1}(w)$ such that $vol_G(B_{k+1}(w)) \leq |E|/2$. By the assumption, it holds that $vol_G(B_{k+1}(w))) \leq vol_G(V \setminus B_{k+1}(w))$, and therefore:

$$\delta_G(B_{k+1}(w)) \geq \phi \cdot vol_G(B_{k+1}(w)) \; . \tag{5.5}$$

By definition of $vol_G(S)$, we have:

$$vol_G(B_{k+1}(w)) \geq vol_G(B_k(w)) + \delta_G(B_{k+1}(w)) \; . \tag{5.6}$$

Combining Eq. (5.5) and Eq. (5.6), we conclude that

$$\begin{aligned} vol_G(B_{k+1}(w)) &\geq vol_G(B_k(w)) + \delta_G(B_{k+1}(w)) \\ &\geq vol_G(B_k(w)) + \phi \cdot vol_G(B_{k+1}(w)) \\ &\geq (1+\phi) \cdot vol_G(B_k(w)) \\ &\geq (1+\phi)^{k+1}, \end{aligned}$$

where the last inequality follows by the induction assumption. Let k be the minimal integer satisfying that $(1+\phi)^k \geq |E|/2$, and consider a fixed pair of vertices u and v. By the above, we have that $|vol_G(B_k(u))|, |vol_G(B_k(v))| \geq |E|/2$. Thus, there is an edge (w_1, w_2) where $w_1 \in B_k(u)$ and $w_2 \in B_k(v)$, and therefore $\text{dist}_G(u,v) \leq 2k+1 = O(\log n/\phi)$. □

By combining Theorem 18 with Fact 7 we get:

Corollary 5. *Let $G = (V,E)$ be an n-vertex ϕ-expander graph with minimum degree $\Theta(f \log n/\phi)$. Let G' be a subgraph of G obtained by sampling each G-edge independently into G' with probability of $p = \Theta(1/f)$. Then G' has diameter of $O(\log n/\phi)$, w.h.p.*

Proof. Since the minimum degree of G is $\Delta = \Theta(f \log n/\phi)$, it holds that $\Delta \geq \kappa \cdot \phi$ for $\kappa = t \log n/\phi^2$, and $p = \Theta(\log n/(\kappa \cdot \phi^2))$. Therefore, by Theorem 18 it holds that $\phi(G') = \Theta(\phi)$, w.h.p. By Fact 7, we have that the diameter of G' is $O(\log n/\phi)$. □

We next show that for *ϕ-expander* graphs with minimum degree $\Theta(f \log n/\phi)$, there exist covering graph families with a considerable smaller cardinality than those obtained for *general* graphs. Note that the definition of the covering family in Lemma 13 is more relaxed than that of Definition 17 w.r.t property (P1).

Lemma 13. *Let G be an n-vertex ϕ-expander graph with minimum degree $\Theta(f \log n/\phi)$. There is a randomized algorithm that computes an $(L, 2f)$ covering family $\mathcal{G}$ of $O(f \cdot \log n)$ subgraphs for $L = O(\log n/\phi)$, satisfying the following properties with high probability. For every $u, v, E' \in V \times V \times E^{\leq 2f}$, there exists a subgraph G_i such that (P1') $\text{dist}_{G_i}(u, v) \leq L$ and (P2) $E' \cap G_i = \emptyset$.*

Proof. The covering family $\mathcal{G}$ is obtained by sampling a collection of $O(f \cdot \log n)$ subgraphs. Specifically, each subgraph $G_i \in \mathcal{G}$ is obtained by sampling each G-edge into G_i independently with probability of $p = \Theta(1/f)$. By Cor. 5, it holds that w.h.p. G_i is a connected graph with diameter $O(\log n/\phi)$. We now show that w.h.p. $\mathcal{G}$ is an $(L, 2f)$ covering family for $L = O(\log n/\phi)$.

Fix a pair of vertices $u, v \in V$ and a subset of at most $2f$ edges $E' \subseteq G$. We claim that with probability of at least $1 - 1/n^{\Omega(f)}$, there exists a subgraph G_i satisfying (P2). The probability that $E' \cap G_i = \emptyset$ is at least $q = (1-p)^{2f} \geq 1/e^3$. Therefore, the probability that no subgraph in $\mathcal{G}$ satisfies (P2) is at most $(1-q)^{c \cdot t \log n} \leq 1/n^{c't}$. By applying the union bound over all $O(n^{4t})$ possible subsets of $2f$ edges, we get that w.h.p. property (P2) holds for every subset E'. In addition, by Cor. 5 w.h.p. it holds that every G_i is connected and has diameter $L = O(\log n/\phi)$. Therefore, w.h.p., for every pair of vertices $u, v \in V$ it holds that $\text{dist}_{G_i}(u, v) \leq L$. By the union bound over the subgraphs in $\mathcal{G}$, we get that w.h.p. both properties (P1') and (P2) hold for every $u, v, E' \in V \times V \times E^{\leq 2f}$. □

Unfortunately, it is unclear how to compute the covering family of Lemma 13 in the adversarial CONGEST model. The reason is that the endpoints of an adversarial edge $e = (u, v) \in F$ cannot faithfully sample e with probability p. For example, letting the endpoint of larger ID u perform the sampling, the other endpoint v might not be correctly informed with the outcome of this sampling[1]. To resolve this, we let each endpoint sample a directed edge (u, v) with a probability of p. Consequently, the graph family that we get consists of directed graphs, and the graph is required to have minimum degree $\Theta(f^2 \log n/\phi)$.

Lemma 14 (Directed Covering Families for Expanders)**.** *For any n-vertex ϕ-expander graph $G = (V, E)$ with minimum degree $\Theta(f^2 \log n/\phi)$, there is a randomized distributed algorithm that in the presence of at most f adversarial edges F, locally computes a* directed *$(L, 2f)$ covering family of $O(f \log n)$* directed *subgraphs $\mathcal{G}$ for $L = O(\log n/\phi)$, satisfying the following w.h.p. For every $u, v, E' \in V \times V \times E^{\leq 2f}$, there exists a subgraph G_i such that (P1') $\text{dist}_{G_i}(u, v) \leq L$ and (P2) $E' \cap G_i = \emptyset$ (i.e., G_i does not contain any edge in E' in neither direction).*

Proof. The covering family $\mathcal{G}$ is obtained by sampling a collection of $O(f \log n)$ *directed* subgraphs. Specifically, each subgraph $G_i \in \mathcal{G}$ is obtained by sampling a directed edge (u, v) with

[1]We note that in the previous constructions, the covering family is constructed locally by all the vertices using Fact 6 (i.e., without any communication). Therefore the above problem is avoided.

probability of $p = \Theta(1/f)$. The sampling of a directed edge (u, v) is done locally by v. We now show that w.h.p. $\mathcal{G}$ is a (directed) $(L, 2f)$ covering family for $L = O(\log n/\phi)$.

Fix a pair of vertices $u, v \in V$ and a subset of at most $2f$ edges $E' \subseteq G$. We claim that with a probability of at least $1 - 1/n^{\Omega(f)}$, there exists a subgraph G_i satisfying (P2). The probability that $E' \cap G_i = \emptyset$ is at least $q = (1-p)^{4t} \geq 1/e^5$. Therefore, the probability that no subgraph in $\mathcal{G}$ satisfies (P2) is at most $(1-q)^{c \cdot t \log n} = 1/n^{c't}$. By applying the union bound over all $O(n^{4t})$ possible sets of $2f$ edges, we get that w.h.p. property (P2) holds for every subset E'.

As each directed edge is sampled with probability p, each edge is sampled in both directions with probability $p^2 = \Theta(1/f^2)$. Since G has minimum degree $\Theta(f^2 \log n/\phi)$, by Cor. 5 it holds that w.h.p. G_i contains a bidirected subgraph which (when viewed as an undirected subgraph) has diameter $O(\log n/\phi)$. As a result, w.h.p. it holds that every G_i has a round-trip diameter $O(\log n/\phi)$. Therefore, for $L = \Theta(\log n/\phi)$ w.h.p. for every pair of vertices $u, v \in V$ it holds that $\text{dist}_{G_i}(u, v) \leq L$. By the union bound over the subgraphs in $\mathcal{G}$, we get that w.h.p. both properties (P1') and (P2) hold for every $u, v, E' \in V \times V \times E^{\leq 2f}$. □

Broadcast algorithm using a directed covering family (Proof of Theorem 14). Let G be an n-vertex ϕ-expander graph, with minimum degree $\Theta(f^2 \log n/\phi)$, and a fix set of f unknown adversarial edges F. Throughout the algorithm, we assume that the vertices hold a linear estimate on the expansion parameter ϕ. (This assumption can also be avoided by adding a logarithmic overhead in the graph diameter to the final round complexity.)
The algorithm begins with locally computing an $(L, 2f)$ directed covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ for G of size $\ell = O(f \log n)$ using Lemma 14, where $L = O(\log n/\phi)$. Next, the vertices execute the broadcast algorithm $\mathsf{BroadcastKnownCovFamily}(L, f)$ of Theorem 17 over $\mathcal{G}$.

Recall that the algorithm of Theorem 17 consists of two phases, a flooding phase where heard bundles are propagated over the subgraphs in the covering family, and an acceptance phase. The flooding phase proceeds in ℓ iterations, where each iteration is implemented in $O(L)$ rounds. In every iteration i, the vertices propagate heard bundles over the directed subgraph $G_i \in \mathcal{G}$. Specifically, a vertex $v \in V$ stores and sends only heard bundles that are received over its *directed* incoming edges, which are sampled (locally by v) into G_i. Every message received by v from a neighbor $u \in N(v)$, such that $(u, v) \notin G_i$ is ignored. The acceptance phase is executed in $O(L)$ rounds, as described in Section 5.3.

Correctness. We now show that the correctness of the algorithm still holds. Let s be the designated source vertex, and let m_0 be the message s sends using the broadcast algorithm. We first note that due to Claim 27, no vertex $v \in V$ accepts a wrong message $m' \neq m_0$. We are left to show all vertices in V accept the message m_0 during the second phase.

Claim 30. *All vertices accept m_0 within $O(L)$ rounds from the beginning of the second phase.*

Proof. We show that all vertices accept the message m_0 by induction on the distance from the source s in the graph $G \setminus F$. The base case holds since s accepts the message m_0 in round

0. Assume all vertices at distance at most i from s in $G \setminus F$ accepts the message by round i. Consider a vertex v at distance $(i+1)$ from s in $G \setminus F$. By the induction assumption on i, the vertex v receives the message $accept(m_0)$ from a neighbor w, in round $j \leq (i+1)$ over a reliable edge (w, v). We are left to show $\text{MinCut}(s, v, \mathcal{P}) \geq t$, where $\mathcal{P}$ is as given by Eq. (5.4):

$$\mathcal{P} = \{P \mid v \text{ stored a } heard(m', len, P) \text{ message and } (v, w), (w, v) \notin P\} .$$

Alternatively, we show that for every edge set $E' \subseteq E \setminus \{(w, v)\}$ of size $(f-1)$, the vertex v stores a heard bundle containing m_0 and a path P_k such that $P_k \cap (E' \cup \{(w, v)\}) = \emptyset$ during the first phase. This necessary implies that the minimum-cut is at least f.

By Lemma 14 there exists a subgraph $G_k \in \mathcal{G}$ satisfying: (P1') $dist_{G_k}(s, v) \leq L$, and (P2) $G_k \cap (F \cup E' \cup \{(v, w)\}) = \emptyset$. Hence, all directed edges in G_k are reliable, and the only message passed through the heard bundles during the k-th iteration is the correct message m_0. Additionally, as G_k contains a directed s-v path of length $O(L)$, the vertex v stores a heard bundle $heard(m_0, x, P_k)$ during the k-th iteration, for some s-v path P_k of length x. As $P_k \subseteq G_k$, it also holds that $P_k \cap (E' \cup \{(v, w)\}) = \emptyset$. We conclude that $\text{MinCut}(s, v, \mathcal{P}) \geq t$, and by the definition of Phase 2, v accepts m_0 by round $(i+1)$. Since $|F| \leq t$, Lemma 14 implies that the diameter of $G \setminus F$ is $O(L)$, and the claim follows. □

Round complexity. The first phase consists of $\ell = O(f \log n)$ iterations, each implemented using $O(L) = O(\log n/\phi)$ rounds. The second phase takes $O(f \log n/\phi)$ rounds. Hence the total round complexity is bounded by $O(f \cdot \log^2 n/\phi)$.

5.5 Improved Broadcast with Shared Randomness

In this section we provide broadcast algorithms with improved bounds provided that the vertices know a shared seed of $\widetilde{O}(1)$ random coins. The structure of this section is as follows. In Section 5.5.1, we present a nearly optimal broadcast algorithm against a single adversarial edge. This algorithm is based on a partial derandomization of the fault-tolerant sampling technique. Then, in Section 5.5.2 we show an improved broadcast algorithm that is resilient against t adversarial edges for $\Omega(t \log n)$ edge-connected expander graphs. This improves upon the broadcast algorithm of Theorem 14 for expander graphs with edge connectivity $\Omega(t^2 \log n)$. Both results are based on providing partial derandomization results for computing covering families. Since these families have many applications in fault-tolerant network design as well dynamic algorithms we believe this contribution to be of independent interest.

5.5.1 Improved Broadcast Algorithm against an Adversarial Edge

Our main technical contribution is in proving the following theorem:

Theorem 19. *Given a graph G where all vertices know a shared seed of $\widetilde{O}(1)$ random bits, there exists a 0-round algorithm for computing an $(L,1)$ covering family $\mathcal{G}=\{G_1,\ldots,G_\ell\}$ for $L=O(D)$, such that $\ell=O(D\log n)$, and the width of $\mathcal{G}$ is $O(\log n)$ satisfying the following properties with high probability. For every $v,e,e'\in V\times V\times E\times E$, there exists a subgraph G_i such that (P1') $\mathrm{dist}_{G_i\setminus\{e,e'\}}(s,v)\le L$ and (P2) $e\notin G_i=\emptyset$.*

Theorem 15 follows by combining Theorem 19 with Theorem 16. We start with describing a standard randomized construction of $\mathcal{G}$ using the fault-tolerant sampling technique. We then turn to prove Theorem 19, by providing a partial derandomization of the randomized construction.

Covering families with fault tolerant sampling. We start by considering a standard randomized construction of $\mathcal{G}$ using the fault-tolerant sampling technique: each subgraph G_i is defined by sampling each edge $e\in G$ into G_i independently with probability of $p=1-1/D$. This is done for $i\in\{1,\ldots,\ell=O(D\log n)\}$. We next analyze this construction.

For a fixed adversarial edge e', the subgraph $G_j\in\mathcal{G}$ is defined to be *good* for the pair $v\in V$ and $e\in E$ if

$$\pi(s,v,G\setminus\{e,e'\})\subseteq G_j \text{ and } e\notin G_j\ . \tag{5.7}$$

Note that since the graph G is 3 edge-connected, by Obs. 6 $\pi(s,v,G\setminus\{e,e'\})$ has at most $7D$ edges for every $v\in V$ and $e\in E$. Thus, the probability that G_j is good for (v,e) is at least $p^{7D}\cdot(1-p)=\Omega(1/D)$. Using Chernoff bound, we get that with w.h.p. every pair (v,e), has at least one good subgraph G_j in $\mathcal{G}$, then $\mathcal{G}$ has the covering property w.h.p. In addition, w.h.p. the width is logarithmic.

Our starting observation is that the covering family $\mathcal{G}$ can be generated using a seed of length $O(D^2\log n)$. To see this, observe that the event of Eq. (5.7) is defined only on $O(D)$ edges, and thus it is sufficient to sample the edges into the G_i subgraphs using D-wise independent hash function. As the sampling is repeated $r=O(D\log n)$ times, this requires a total random seed of length $O(D^2\log n)$ bits.

(Partial) Derandomization of the FT-Sampling Technique

In this section, we considerably improve this bound and prove Theorem 19 by showing that one can compute $\mathcal{G}$ using a random seed of $\widetilde{O}(1)$ bits. The structure of our argument is as follows. We first show that a single iteration of the sampling procedure can be implemented using $r=O(\log n(\log\log n)^3)$ random bits. That is, we show that if all vertices share a seed of r random bits, then for every fixed pair e,e' the sampled graph obtained using this seed satisfies the event of Eq. (5.7) with probability $\Omega(1/D)$. This immediately implies that the entire algorithm, which consists of $O(D\log n)$ sampling steps, can be implemented with $\widetilde{O}(D)$ bits. In the next step, we derandomize the algorithm even further and show that using $O(\log n)$-wise independent hash functions, one can simulate the entire construction of $\mathcal{G}$ with a seed of length $\mathsf{poly}(\log n)$.

Single subgraph with a seed length $O(\log n(\log\log n)^3)$. Our key observation is that for any pair $\langle A, e'\rangle$ where $A \subseteq E$, the event that all edges A are sampled into G_i and $e' \notin G_i$ can be expressed by a *read-once CNF formula*[1]. Thus, in order to get a short seed, it suffices to have a pseudorandom generator (PRG) that can "fool" read-once CNFs. A PRG is a function that gets a short random seed and expands it to a long one which is indistinguishable from a random seed of the same length for such a formula. Gopalan et al. [80] showed such a PRG that have a seed length $O(\log n \cdot (\log\log n)^3)$. Our application of their PRGs follows a similar line to that of [153].

We begin by setting up some notation. For a function PRG and an index i, let $\mathsf{PRG}(s)_i$ be the i-th bit of $\mathsf{PRG}(s)$.

Definition 21 (Pseudorandom Generators for Boolean Functions). *A generator* $\mathsf{PRG}\colon \{0,1\}^r \to \{0,1\}^n$ *is an ϵ-pseudorandom generator (PRG) for a class $\mathcal{C}$ of Boolean functions if for every $f \in \mathcal{C}$:*

$$|\mathop{\mathbf{E}}_{x\sim\{0,1\}^n}[f(x)] - \mathop{\mathbf{E}}_{s\sim\{0,1\}^r}[f(\mathsf{PRG}(s))]| \le \epsilon.$$

We refer to r as the seed-length of the generator and say PRG *is explicit if there is an efficient algorithm to compute it that runs in time* $\mathsf{poly}(n, 1/\epsilon)$.

We use the construction of [80] to provide a PRG for CNFs with a short seed.

Theorem 20 ([80]). *For every $\epsilon = \epsilon(n) > 0$, there exists an explicit pseudorandom generator,* $\mathsf{PRG}\colon \{0,1\}^r \to \{0,1\}^n$ *that fools all read-once CNFs on n-variables with error at most ϵ and seed-length $r = O((\log(n/\epsilon)) \cdot (\log\log(n/\epsilon))^3)$.*

Using the notation above, and Theorem 20 we formulate and prove the following Lemma:

Lemma 15. *Let G be an n-vertex graph, and m be an integer where $m = \mathsf{poly}(n)$. For every $D \le n$, there exists a family of hash functions $\mathcal{H} = \{h\colon [m] \to \{0,1\}\}$ such that choosing a random function from $\mathcal{H}$ takes $r = O(\log n \cdot (\log\log n)^3)$ random bits, such that for $Z_h = \{e \in [m] : h(e) = 0\}$ and any fixed pair of $O(D)$ edges A and an edge $e' \in E$ it holds that:*

$$\Pr_h[A \subset Z_h \text{ and } e' \notin Z_h] \ge \Omega(1/D) - 1/n^2 \text{ and } \Pr_h[e \notin Z_h] \le O(1/D) + 1/n^2 \ \forall e \in E.$$

Proof. We first describe the construction of $\mathcal{H}$. Let $p = 1 - c/D$ for some large constant c, and let $\ell = \lfloor \log 1/p \rfloor$. Let $\mathsf{PRG}\colon \{0,1\}^r \to \{0,1\}^{m\ell}$ be the PRG constructed in Theorem 20 for $r = O(\log n\ell \cdot (\log\log n\ell)^3) = O(\log n \cdot (\log\log n)^3)$ and for $\epsilon = 1/n^{10c}$. For a string s of length r we define the hash function $h_s(j)$ as follows. First, it computes $y = \mathsf{PRG}(s)$. Then, it interprets y as m blocks where each block is of length ℓ bits, and outputs 1 if and only if all the bits of the j-th block are 1. Formally, we define $h_s(j) = \bigwedge_{k=(j-1)\ell+1}^{j\ell} \mathsf{PRG}(s)_k$. We show that the requirement holds for the set Z_{h_s} where $h_s \in \mathcal{H}$ and a fixed set of $O(D)$ edges $A \subset E$ and

[1] A read-once formula is a Boolean formula in which each variable occurs at most once.

$e' \in E \setminus A$. For $j \in [m]$ let $X_j = h_s(j)$ be a random variable where $s \sim \{0,1\}^r$. Let the IDs of the edges in A be given by $j_1, \ldots, j_k$ and let j^* be the ID of the edge e'. We need to show that $\Pr_{s \sim \{0,1\}^r}[X_{j_1} \wedge X_{j_2} \wedge \ldots \wedge X_{j_k} \wedge \neg X_{j^*} = 1] \geq 1/D + 1/n^2$. Let $f_j \colon \{0,1\}^{m\ell} \to \{0,1\}$ be a function that outputs 1 if the j-th block is all 1's. That is, $f_j(y) = \bigwedge_{z=(j-1)\ell+1}^{j\ell} y_j$. Since f_j is a read-once CNF formula we have that

$$\left| \mathop{\mathbf{E}}_{y \sim \{0,1\}^{m\ell}}[f_j(y)] - \mathop{\mathbf{E}}_{s \sim \{0,1\}^r}[f_j(\mathsf{PRG}(s))] \right| \leq \epsilon.$$

Thus, the event where $e \in Z_h$ or more generally, $A \subset Z_h$ can be written as a read-once CNF formula. In addition, the event where $e' \notin Z_h$ is obtained in case where at least one of the ℓ bits of the j^*-th block is zero, which can be expressed as a CNF formula as well. Over all, we have that the good event where $A \subset Z_h$ **and** $e' \notin Z_h$ is a read-once CNF formula, and $\Pr_h[A \subset Z_h \text{ and } e' \notin Z_h] \geq \Omega(1/D) - 1/\epsilon$. The claim follows by setting $\epsilon = 1/\mathsf{poly}(n)$. $\square$

We next describe the covering family algorithm when using a random seed of length $r = \widetilde{O}(D)$. The algorithm has $\ell = O(D \log n)$ iterations. In each iteration i, let $s_i \sim \{0,1\}^r$ be a random seed shared by all vertices. The vertices then locally interpret the vector $y_i = \mathsf{PRG}(s_i)$ as m blocks, one per possible edge in G, where each block is of length ℓ bits. The vector y_i defines the subgraph G_i as follows. For every edge $e = (u,v)$ where $ID(u) < ID(v)$ let $ID_{u,v}$ be an $O(\log n)$-bit ID of the edge obtained by concatenating[1] the IDs of u and v. Then, to decide whether $e = (u,v) \in G_i$, the vertices u and v consider the $2^{ID_{u,v}}$-th block of the vector y_i, and the edge e joins G_i iff *all* the bits of this block are set to 1. Lemma 15 then implies the following:

Corollary 6. *The probability that Eq. (5.7) holds for a given tuple $\langle s, v, \{e, e'\} \rangle$ based on the above definition of G_i is $\Omega(1/D)$. In addition, the probability that $e \notin G_i$ is bounded by $O(1/D)$.*

Therefore, by applying ℓ independent repetitions, we get that each edge pair e, e' has a good iteration with high probability. In addition, by Chernoff bound, each edge e does not appear on at most $\widetilde{O}(1)$ subgraphs.

Sampling $\widetilde{O}(D)$ subgraphs with a seed length $\widetilde{O}(1)$. So-far, we showed that a sampling of a single subgraph G_j can be implemented with a random seed of $\ell = \log n \cdot \mathsf{poly}(\log\log n)$ bits. That is, the output sampled subgraph satisfies Eq. (5.7) with probability of $\Omega(1/D)$. Our goal now is to sample $\ell = O(D \log n)$ subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ that satisfy the guarantee of Theorem 19 using a seed of $\widetilde{O}(1)$.

We next show that it is sufficient for the PRG's seeds used in the sampling of the G_i subgrpahs to be only $O(\log n)$-wise independent rather than fully independent. This would imply that all $\ell = O(D \log n)$ subgraphs can be sampled with a seed of poly-logarithm length.

[1]To make this ID consistent, the ID of the lower-ID endpoint appears first in this concatenation.

Definition 22 ([188, Definition 3.31]). *For $N, M, d \in \mathbb{N}$ such that $d \leq N$, a family of functions $\mathcal{H} = h : [N] \to [M]$ is d-wise independent if for all distinct $x_1, x_2, ..., x_d \in [N]$, the random variables $H(x_1), ..., H(x_d)$ are independent and uniformly distributed in $[M]$ when H is chosen randomly from $\mathcal{H}$.*

In [188] an explicit construction of $\mathcal{H}$ is presented, with the following parameters.

Lemma 16 ([188, Corollary 3.34]). *For every $\gamma, \beta, d \in \mathbb{N}$, there is a family of d-wise independent functions $\mathcal{H}_{\gamma,r} = h : \{0,1\}^\gamma \to \{0,1\}^r$ such that choosing a random function from $\mathcal{H}_{\gamma,r}$ takes $d \cdot \max\{\gamma, r\}$ random bits, and evaluating a function from $\mathcal{H}_{\gamma,r}$ takes time $\mathsf{poly}(\gamma, r, d)$.*

Our construction is based on the family of d-wise independent functions $\mathcal{H}_{\gamma,\beta}$ of Lemma 16 with $d = O(\log n)$, $\gamma = O(\log n)$ and $r = O(\log n \cdot \log\log^3 n)$. By Lemma 16, choosing a random function from $\mathcal{H}_{\gamma,r}$ can be done with a random seed of length $a = O(\log^2 n \cdot \log\log^3 n)$. Let $S \sim \{0,1\}^a$ be a random seed, and let h_S be the random function chosen from $\mathcal{H}_{\gamma,r}$ using S.

For every $i \in \{1, \ldots, \ell\}$, the vertices will be using the r-length seed $h_s(i) \in \{0,1\}^r$ to define the subgraph G_i as follows. The r-length seed $h_S(i) \in \{0,1\}^r$ is fed into the PRG function $\mathsf{PRG}\colon \{0,1\}^r \to \{0,1\}^{m\ell}$ of Lemma 15 where $\ell = O(\log n)$. The $m\ell$-length bit output vector $y = \mathsf{PRG}(h_S(i))$ is interpreted by the vertices as m blocks, where each block is of length ℓ bits. Every edge (u, v) has an ID of size $O(\log n)$ bits $ID_{u,v}$ (w.l.o.g., $ID(u) < ID(v)$). For every edge $e = (u, v)$, the seed s is interpreted as follows:

$$e \in G_i \text{ iff all bits of the } i\text{-th block in } y = \mathsf{PRG}(h_S(i)) \text{ are } 1 \text{ where } i = 2^{ID_{u,v}}\,. \tag{5.8}$$

Using this shared seed S, each vertex u can decide locally whether each of its incident edges belong to G_i. Importantly, this decision is *consistent* between the edge endpoints without using any *communication*. The algorithm will repeat this for $O(D \log^2 n)$ iterations. We next show that when the G_i's are defined based on Eq. (5.8) every tuple $\langle s, v, \{e, e'\}\rangle$ has a good iteration w.h.p. We will use the following fact.

Fact 8. *[Theorem 2.3 [20]] Let $X_1, \ldots, X_n \in \{0,1\}$ be $2k$-wise random variables for some $k \in \mathbb{N}_{\geq 2}$, and let $X = \sum_{i=1}^n X_i$ and $\mu = \mathbf{E}[X]$. It holds that $\Pr[|X - \mu| \geq A] \leq c \cdot ((2k \cdot \mu + (2k)^2)/A^2)^k$ for some positive constant $c \leq 8$.*

Lemma 17. *Using a seed of $\widetilde{O}(1)$-bits as explained above for $\tau = O(D \log^2 n)$ iterations, then for every tuple $\langle s, v, \{e, e'\}\rangle$, there exists a good iteration (i.e., satisfying Eq. (5.7)) with high probability.*

Proof. Fix $S \sim \{0,1\}^a$ for $a = O(\log^2 n \cdot \log\log^3 n)$ and a single iteration i. By the definition of the d-wise independent hash function family $\mathcal{H}_{\beta,r}$, when choosing a function $h \in \mathcal{H}_{\beta,r}$ with a seed $S \sim \{0,1\}^a$, all bits of $h_S(i)$ are uniformly distributed in $\{0,1\}^r$. Thus $h_S(i) \sim \{0,1\}^r$ and by applying the argument of Lemma 15 on the graph G_i (defined based on Eq. (5.8)), we get that the event of Eq. (5.7) holds with probability at least $10/D$. We next show that w.h.p. at least one of the $\tau = 20D\log^2 n$ iterations is good for $\langle s, v, \{e, e'\}\rangle$.

Let X_i be the indicator random variable for the event that Eq. (5.7) holds in iteration i when using $S \sim \{0,1\}^a$ to define the sub-graph G_i (see Eq. (5.8)). Let $X = \sum_{i=1}^{\ell} X_i$ and $\mu = \mathbf{E}_{S\sim\{0,1\}^a}[X]$. By the linearity of expectation, $\mu \geq 2\log^2 n$. By definition, the random variables $h_S(1), \ldots, h_S(t)$ are $10\log n$-wise independent. Since each X_i is a random variable that depends only on the bits of $h_S(i)$, we get that the X_i's variables are $10\log n$-wise independent. By using the concentration inequality of Fact 8 with $k = 5\log n$, we get that

$$\Pr\left[|X-\mu| > \log^2 n\right] \leq c \cdot ((20\log^3 n + 100\log^2 n)/\log^4 n)^{\log n} \leq c/n^{10}\ .$$

The claim holds by applying the union bound over all n^2 pairs. The same argument holds for the width as well. □

This completes the proof of Theorem 19.

5.5.2 Improved Broadcast Algorithm for Expander Graphs

We next show that using a shared seed of $O(\log n)$ bits, all vertices can compute locally a covering family with improved properties. This consequently leads to an improved broadcast algorithm that is resilient against t adversarial edges, given that G is a ϕ-expander graph with edge connectivity $\Omega(t\log n/\phi)$. Given a seed of $O(\log n)$ random bits, the computation of the improved covering family is performed in 0 rounds of communication, but requires an exponentially large local time computation at each vertex.

Lemma 18. *Let G be an n-vertex ϕ-expander graph with edge connectivity $\Omega(t\log n/\phi)$. Given a shared seed S of $O(\log n)$ random bits, there is a 0-round algorithm for computing an $(L, 2t)$-covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ of $\ell = O(t \cdot \log n)$ subgraphs for $L = O(\log n/\phi)$ where properties (P1') and (P2) of Lemma 13 hold high high probability. Given the seed S, each vertex v can (locally) determine its incident edges in each $G_i \in \mathcal{G}$. The local computation at each vertex is exponential.*

The proof of Lemma 18 follows by using the brute-force construction of PRGs.

Definition 23 (Computational Indistinguishability, Definition 7.1 in [188]**).** *Random variables X and Y taking values in $\{0,1\}^m$ are (r,ϵ)* indistinguishable *if for every nonuniform algorithm T running in time at most r, we have $|\Pr[T(X)=1] - \Pr[T(Y)=1]| \leq \epsilon$.*

Definition 24 (PRGs for T-Time Algorithms, Definition 7.3 in [188])**.** *A deterministic function $\mathcal{G} : \{0,1\}^d \to \{0,1\}^m$ is an (r,ϵ) pseudorandom generator (PRG) if: (1) $d \leq m$ and (2) $\mathcal{G}(U_d)$ and U_m are (r,ϵ) indistinguishable, where U_ℓ is a uniform sample of ℓ random bits for $\ell \in \{d, m\}$.*

Using the probabilistic method, one can show that the family of polynomial time randomized algorithms can be simulated with a seed of $O(\log n)$ random bits with a polynomially small error.

Proposition 2 (Proposition 7.8 in [188])**.** *For all $m \in \mathbb{N}$ and $\epsilon > 0$, there exists a (non-explicit) (m, ϵ) PRG $\mathcal{PR} : \{0,1\}^d \to \{0,1\}^m$ with seed length $d = O(\log m + \log 1/\epsilon)$.*

We are now ready to complete the proof of Lemma 18.

Proof of Lemma 18. Given a seed S of logarithmic length $d = O(\log n)$, the vertices can locally compute the PRG $\mathcal{PR} : \{0,1\}^d \to \{0,1\}^m$ for $m = \mathsf{poly}(n)$, that ϵ-fools the sampling of the covering family $\mathcal{G}$ from Lemma 13 for $\epsilon = 1/\log n$. The $\mathsf{poly}(n)$ bits $\mathcal{PR}(S)$ consists of $N = \mathsf{poly}(n)$ chunks, each chunk i is of size $|\mathcal{G}| \cdot O(\log n)$ which specifies the pseudorandom coins for simulating the sampling of the i-th edge e_i into each of the $\mathcal{G}$'s subgraphs, where N is the polynomial estimate of the vertices on the number of edges in the graph. $\square$

The proof of Lemma 11 follows from combining Lemma 18 and Theorem 17.

6 General CONGEST Compilers against Adversarial Edges

6.1 Introduction

As communication networks grow in size, they become increasingly vulnerable to failures and byzantine attacks. It is therefore crucial to develop fault-tolerant distributed algorithms that work correctly despite the existence of such failures, without knowing their location. The area of fault-tolerant distributed computation has attracted a lot of attention over the years, especially since the introduction of the byzantine agreement problem by Pease, Shostak and Lamport [159, 109]. The vast majority of these algorithms, however, assume that the communication graph is the complete graph [57, 58, 65, 34, 184, 169, 25, 170, 24, 63, 75, 69, 107, 162, 103, 59, 137, 92, 47, 105]. For the latter, one can provide time efficient algorithms for various distributed tasks that can tolerate up to a *constant* fraction of corrupted edges and vertices [57, 23, 25, 59]. Very little is known on the complexity of fault-tolerant computation for general graph topologies. In a seminal work, Dolev [57] showed that any given graph can tolerate up to f adversarial vertices iff it is $(2f+1)$ vertex-connected. Unfortunately, the existing distributed algorithms for general $(2f+1)$ connected graphs usually require a polynomial number of rounds in the CONGEST model of distributed computing [164].

In this work, we present a general compiler that translates any given distributed algorithm $\mathcal{A}$ (in the fault-free setting) into an equivalent algorithm $\mathcal{A}'$ that performs the same computation in the presence of f adversarial edges. Our primary objective is to minimize the **compilation overhead**, namely, the ratio between[1] the round complexities of the algorithms $\mathcal{A}'$ and $\mathcal{A}$. We take the gradual approach of fault-tolerant network design, and consider first the case of a single adversarial edge, and later on the case of multiple adversarial edges. We note that, in general, such compilers might not be obtained for adversarial vertices[2] and thus, we focus on edges.

6.1.1 Model Definition and the State of the Art

We consider the adversarial CONGEST model, defined in Chapter 4, where we focus on $(2f+1)$ edge-connected graphs, which can tolerate up to f adversarial edges. The problem of devising general round-by-round compilers in the adversarial CONGEST model boils down into the following distributed task:

Single round compilation in the adversarial CONGEST model: Given is a $(2f+1)$ edge-connected graph $G = (V, E)$ with a fixed set $F^* \subseteq G$ of at most f adversarial edges. Let $\mathcal{M} = \{M_{u\to v} \mid (u, v) \in E\}$ be a collection of $O(\log n)$-bit messages that are required to be sent over (potentially) all graph edges. I.e., for each (directed) edge (u, v), the vertex u has a designated $O(\log n)$-bit message for v.

The single round compilation algorithm is required to exchange these messages in the adversarial CONGEST model, such that at the end of the algorithm, each vertex v holds the correct message $M_{u\to v}$ for each of its neighbors u, while ignoring all remaining (corrupted) messages.

The main complexity measure is the round complexity of the single-round compilation algorithm, which corresponds to the *compilation overhead* of the compiler. The compilation of CONGEST algorithms under various adversarial settings has been recently studied by [155]. We next explain their methodology and discuss our contribution with respect to the state-of-the-art.

The simulation methodology of [155]. Motivated by various applications for *resilient distributed computing*, Parter and Yogev [155] introduced the notion of *low-congestion cycle covers* as a basic communication backbone for reliable communication. Formally, a (c, d)-cycle cover of a two edge-connected graph G is a collection of cycles in G in which each cycle is of length at most d, and each edge participates in at least one cycle and at most c cycles. The *quality* of the cycle cover is measured by $c + d$. Using the beautiful result of Leighton, Maggs and Rao [113] and the follow-up by Ghaffari [78], a (c, d)-cycle cover allows one to route $O(\log n)$ bits of information on all cycles simultaneously in $\widetilde{O}(c + d)$ CONGEST rounds.

[1]Note that we use the term compilation overhead to measure the time it takes to simulate a single fault-free round of algorithm $\mathcal{A}$ in the adversarial setting. This should not be confused with the time required to set up the compiler machinery (e.g., computing the cycle cover).

[2]Such compilers might still be obtained under the stronger KT2 model where vertices know their two-hop neighbors.

Low-congestion cycle covers with parameters c, d give raise to a *simulation* methodology that transforms any distributed algorithm $\mathcal{A}$ and compile it into a *resilient* one; the compilation overhead is $g(c, d)$, for some function g. The resilient simulation exploits the fact that a cycle covering an edge $e = (u, v)$ provides *two*-edge-disjoint paths for exchanging messages from u to v. Parter and Yogev [155] showed that any n-vertex two edge-connected graph with diameter D has a (c, d)-cycle covers with $c = O(1)$ and $d = \widetilde{O}(D)$. These bounds are existentially tight. [156, 157] also presented an r-round CONGEST algorithm for computing (c, d) cycles covers for $r, d = \widehat{O}(D)$ and $c = \widehat{O}(1)$.

Our simulation methodology in the adversarial CONGEST model extends the work of [155] in several aspects. First, the cycle covers of [155] are limited to handle at most *one* edge corruption. To accommodate a large number of adversarial edges, we introduce the notion of *fault-tolerant (FT) cycle cover* which extends low-congestion cycle cover to handle multiple adversarial edges. Informally, a FT cycle cover with parameters c, d is a cycle collection $\mathcal{C}$ that covers each edge e by multiple cycles (instead of one). For every sequence of at most f faults F, there is a cycle C in $\mathcal{C}$ that covers[1] e without visiting any of the edges in $F \setminus \{e\}$. All cycles in $\mathcal{C}$ are required to be of length at most d, and with an overlap of at most c, to allow an efficient information exchange over all these cycles in parallel.

A key limitation of the compilers provided by [155] is that they assume the cycle covers themselves are computed in a (fault-free) preprocessing phase. These cycles are then used by the compilers in the adversarial CONGEST model. Our main goal in this work is to omit this assumption and provide efficient algorithms for computing FT cycle covers in the adversarial CONGEST model. The computation of these cycles in the presence of the adversarial edges is quite intricate. The key challenge is in computing cycles for covering the adversarial edges themselves. The latter task requires some coordination between the endpoints of the adversarial edges, which seems to be quite hard to achieve. Note that the covering of the adversarial edges by cycles is crucial for the compilation task to reliably simulate the message exchange over these edges in the given fault-free algorithm $\mathcal{A}$. Upon computing a FT cycle cover with parameters c, d, we then present a round-by-round compiler whose overhead depends on the c, d parameters. To optimize the round overhead, we exploit (our modified) FT cycle cover in a somewhat more delicate manner compared to that of [155], leading to an improvement by a factor of $O(D^2)$ rounds for a single adversarial edge.

6.1.2 Contributions and Key Results

We consider the design of compilers that can simulate every given distributed algorithm in the adversarial CONGEST model. The compilers are based on a new notion of *FT cycle cover*, an extension of the low-congestion cycle cover [156] to the adversarial setting. We also provide

[1] A stricter requirement is to cover each edge by f edge-disjoint cycles. However, this definition leads to a larger compilation overhead compared to the one obtained with our definition.

a new method to compile an algorithm given the FT cycle cover. We start by describing our contribution w.r.t the combinatorial characterization of FT cycle covers, and then consider the computational aspects in the adversarial CONGEST model.

6.1.2.1 Combinatorial Properties of Fault Tolerant Cycle Covers

We provide first the standard definition of low congestion cycle covers of [155], and then introduce their extension to the fault-tolerant setting. A (c, d) *low-congestion cycle cover* of a two edge-connected graph G is a collection of cycles in G in which (i) each cycle is of length at most d (dilation), and (ii) each edge participates in at least *one* cycle (covering), and at most c cycles (congestion). The *quality* of the cycle cover is measured by $c + d$. In order to provide reliable computation in the presence of f adversarial edges F^*, it is desired to cover each edge by multiple short cycles of small overlap. This motivates the following definition.

Definition 25 (f-FT Cycle Cover)**.** *Given an $(f + 1)$ edge-connected graph G an f-FT cycle cover with parameters (c, d) is a collection of cycles $\mathcal{C}$ such that for any set $E' \subseteq E$ of size $(f - 1)$ and every edge $e \in E$, there exists a cycle $C \in \mathcal{C}$ such that $C \cap (E' \cup \{e\}) = \{e\}$. The length of every cycle in $\mathcal{C}$ is at most d, and each edge participates in at most c cycles.*

In other words, the f-FT cycle cover $\mathcal{C}$ provides for each edge $e = (u, v)$ a subgraph G'_e (consisting of all cycles covering e), such that the minimum u-v cut in G'_e is at least $f + 1$. Using the FT sampling technique from [193, 56], in the full paper [88] we show the following.

Lemma 19 (Upper bound on FT Cycle-Covers)**.** *For every $(f + 1)$ edge-connected graph G with diameter D, there is a randomized construction for computing f-FT cycle cover $\mathcal{C}$ with parameters (c, d) where $c = \widehat{O}(f(5fD)^f)$ and $d = \widehat{O}(fD)$.*

One of our technical contributions is an almost *matching* lower bound for the quality of FT cycle covers. This is done by a careful analysis of the congestion and dilation parameters of replacement paths in faulty graphs. We believe that the following graph theoretical theorem should be of independent interest in the context of fault-tolerant network design and distributed minimum cut computation.

Theorem 21 (Lower Bound on the Quality of FT Cycle Covers)**.** *For every $f \geq 1$, $D \geq f$ and $n = \omega(D^f)$, there exists an n-vertex $(f + 1)$ edge-connected graph $G^* = (V, E)$ with diameter D, such that any f-FT cycle cover with parameters c, d must satisfy that $c + d = (D/f)^{\Omega(f)}$.*

This theorem provides an explanation for the compilation overhead of $D^{\Omega(f)}$ of our compilers. It also provides an explanation for the natural barrier of $D^{\Omega(f)}$ rounds for handling f adversarial edges in the distributed setting. Specifically, the lower bound implies that there exists at least one pair of vertices u, v in the graph G^* such that for any selection of $(f + 1)$ edge-disjoint u-v paths $\mathcal{P}$ in G^*, the longest path in $\mathcal{P}$ must have a length of $(D/f)^{\Omega(f)}$ edges.

Theorem 21 also proves that the collection of all $V \times V \times E^f$ replacement paths[1] avoiding f faults, obtained by the FT sampling technique, are optimal in terms of their congestion + dilation bounds. It also shows that the analysis of the distributed minimum cut algorithm of [152] is nearly *optimal*[2].

A relaxed notion of FT cycle covers. In a setting where a fixed set of edges F^* are adversarial for $|F^*| = f$, it might not be possible to compute a $(2f)$-FT cycle cover as defined by Definition 25. This is despite the fact that we require the edge connectivity of the graph to be at least $(2f+1)$. To see this, consider the scenario where the adversarial edges F^* are completely idle throughout the distributed computation. In such a case, the communication graph becomes $G \setminus F^*$, which is no longer guaranteed to have an edge-connectivity of $(2f+1)$. For this reason, we consider a more relaxed notion of FT cycle covers, that on the one hand, can be computed in the adversarial setting, and on the other hand, is strong enough for our compilers.

Definition 26 ((f, F^*)-FT Cycle Cover)**.** *Given an $(2f+1)$ edge-connected graph G, and a fixed set of unknown adversarial edges $F^* \subseteq E$ of size at most f, an (f, F^*)-FT cycle cover with parameters (c, d) is a collection of cycles $\mathcal{C}$ such that for every edge $e \in E$ (possibly $e \in F^*$), and every set $E' \subseteq E$ of size $|E'| \leq f-1$, there exists a cycle $C \in \mathcal{C}$ such that $C \cap (E' \cup F^* \cup \{e\}) = \{e\}$. The length of each cycle is bounded by d, and every edge appears on at most c cycles in $\mathcal{C}$.*

Note that for every $F \subseteq E$, $|F| \leq f$, an (f, F)-FT cycle cover contains an f-FT cycle cover, and therefore the lower bound of Theorem 21 also holds for (f, F^*)-FT cycle covers. When $F^* = \{e'\}$, we slightly abuse notation and simply write (f, e')-FT cycle cover. Our FT cycle covers should be useful for many other adversarial settings. Specifically, they provide an immediate extension of the compilers of [155] to handle adversaries that corrupt multiple edges, such as eavesdroppers [155] and semi-honest adversaries [156].

We next turn to consider the computational aspects of FT cycle covers, and their applications. In the distributed setting, we assume throughout that the vertices of the graph obtain a linear estimate[3] on the diameter of the graph D. This assumption (also applied in, e.g., [38]) is needed as the compilation overhead is a function of D.

6.1.2.2 Handling a Single Adversarial Edge

We start by considering an adversarial setting with a *single* fixed unknown adversarial edge e'. At the heart of the compiler lies an efficient construction of a $(1, e')$-FT cycle cover in the adversarial CONGEST model.

[1] A replacement path is a shortest path in some graph $G \setminus F$.

[2] This algorithm computes the minimum cut by computing for each vertex v the collection of all replacement paths w.r.t a fixed source vertex s.

[3] This assumption can be omitted using the broadcast algorithms of Chapter 5, in the case where the vertices have a designated marked leader.

Theorem 22 ($(1, e')$-FT Cycle Cover)**.** *Consider a* 3 *edge-connected* n*-vertex graph* G *of diameter* D*, and a fixed adversarial edge* e'*.*

- *There is an* r*-round deterministic algorithm for computing a* $(1, e')$*-FT cycle cover with congestion and dilation* $c = \widehat{O}(D^2), d = \widehat{O}(D)$*, and* $r = \widehat{O}(D^4)$ *in the adversarial* CONGEST *model.*
- *There is an* r*-round randomized algorithm for computing* $(1, e')$*-FT cycle cover, w.h.p., with congestion and dilation* $c, d = \widehat{O}(D)$ *and* $r = \widehat{O}(D^2)$ *in the adversarial* CONGEST *model.*

In the distributed output format of the $(1, e')$-FT cycle cover computation, the endpoints of every edge $e = (u, v)$ hold the unique identifiers of all the cycles $\mathcal{C}_e$ covering e, as well as, a full description of these cycles. The key challenge in proving Theorem 22 is in covering the adversarial edge e'. For that purpose, we provide a delicate cycle verification procedure that allows the endpoints of each edge $e = (u, v)$ to correctly identify if e is currently covered by a (legal) cycle. This verification is robust to the behavior of the adversarial edge. Using these cycle covers, we obtain general compilers against e'.

Theorem 23. *(Compiler against a Single Adversarial Edge) Given is a* 3 *edge-connected* D*–diameter graph* G *with a fixed adversarial edge* e'*, and a* $(1, e')$*-FT cycle cover* $\mathcal{C}$ *with parameters* (d, c) *for* G *(e.g., as obtained by Theorem 22). Then any distributed algorithm* $\mathcal{A}$ *can be compiled into an equivalent algorithm* $\mathcal{A}'$ *against* e' *with an overhead of* $O(c \cdot d^2)$ *rounds (in the adversarial* CONGEST *model).*

This improves the compilation overhead of Parter and Yogev [155] by a factor of $\widetilde{O}(D^2)$ rounds. The compilers of [155] are based on exchanging the $M_{u \to v}$ messages of Alg. $\mathcal{A}$ along 3 edge-disjoint u-v paths. In our compilation scheme, instead of insisting on edge-disjoint paths, the messages are exchanged over a collection u-v paths of a sufficiently large *flow*. This leads to improvement in the compilation overhead.

6.1.2.3 Handling Multiple Adversarial Edges

We next consider $(2f + 1)$ edge-connected graphs of diameter D with a fixed set $F^* \subseteq E$ of adversarial edges, $|F^*| \leq f$. To handle f adversarial edges F^* in $(2f + 1)$ edge-connected graphs, we use the notion of (f, F^*)-FT cycle covers. Our first contribution is the construction of (f, F^*)-FT cycle covers in the adversarial CONGEST model. Due to technicalities that arise in this adversarial setting, our final output contains the desired cycles required by (f, F^*)-FT cycle cover, but might include, in addition, also truncated paths which are quite “harmless” in the compilation process later on. Formally, our distributed construction computes a (f, F^*)-FT cycle cover* where the asterisk indicates the possible existence of truncated paths in the distributed output.

Definition 27 ((f, F^*)-FT Cycle Cover*)**.** *Given a $(2f+1)$ edge-connected graph G and a fixed set of adversarial edges $F^* \subseteq E$ where $|F^*| \leq f$, a (f, F^*)-FT cycle cover* with parameters (c, d), is a collection of cycles and paths $\mathcal{C}$ such that $\mathcal{C}$* contains *a (f, F^*)-FT cycle cover for G. The length of each cycle and path in $\mathcal{C}$ is at most d and every edge $e \in E$ appears in at most c cycles and paths.*

Theorem 24 ((f, F^*)-FT Cycle Cover*)**.** *Let G be a $(2f+1)$ edge-connected graph G of diameter D, and a fixed set of f adversarial edges F^*. Then, there exists an r-round deterministic algorithm, in the adversarial* $\mathsf{CONGEST}$ *model for computing a (f, F^*)-FT cycle cover* for G, with parameters $d = \widehat{O}(f \cdot D)$ and $r, c = \widehat{O}((Df \log n)^{O(f)})$.*

Note that by the lower bound result of Theorem 21, the quality of the FT cycle covers must be $(D/f)^{\Omega(f)}$.

Given a (f, F^*)-FT cycle cover* for a graph G, we extend the general compiler of Theorem 23 to handle f adversarial edges.

Theorem 25 (Compilers against f Adversarial Edges)**.** *Given a $(2f+1)$ edge-connected D–diameter graph G with a fixed set of f adversarial edges F^*, and a (f, F^*)-FT cycle cover* with parameters (d, c) for G. Then any distributed algorithm $\mathcal{A}$ can be compiled into an equivalent algorithm $\mathcal{A}'$ against F^*, with a compilation overhead of $O(c \cdot d^3)$ rounds.*

The high level intuitive idea of our compiler is as follows. Fix a round i of Alg. $\mathcal{A}$, and consider the message $M_{u \to v}$ sent over the edge (u, v) in that round. Our compiler lets u send the message $M_{u \to v}$ through all cycles covering e in the (f, F^*)-FT cycle cover*. The vertex v can then recover $M_{u \to v}$ by exploiting the following property. On the one hand, the (f, F^*)-FT cycle cover* covers e by sufficiently many cycles that avoid $F^* \setminus \{e\}$. Consequently, the correct message $M_{u \to v}$ is received by v over a path collection with a u-v flow[1] of at least $f + 1$. On the other hand, any corrupted message $M' \neq M_{u \to v}$ must be propagated along a walk that contains at least *one* adversarial edge. Consequently, a corrupted message M' is propagated over a walk collection with a u-v flow of at most f.

6.1.3 Basic Tools

We consider the adversarial $\mathsf{CONGEST}$ model, defined in Chapter 4. Our distributed algorithms use a graph structures called neighborhood covers defined as follows.

Definition 28 (Neighborhood Covers [13])**.** *The r-neighborhood cover of the graph G is a collection of vertex subsets, denoted as, clusters $\mathcal{N} = \{S_1, \ldots, S_\ell\}$ where $S_i \subseteq V$ such that: (i) every vertex v has a cluster that contains its entire r-radius neighborhood in G, (ii) the diameter of each $G[S_i]$ is $O(r \log^c n)$ for some constant c, and (iii) every vertex belongs to $\widetilde{O}(1)$ clusters in $\mathcal{N}$.*

[1]To formalize this argument, we provide a formal definition for the cut value of a u-v walk collection.

We use the deterministic construction of neighborhood covers by Rohzon and Ghaffari [168] .

Theorem 26 (Corollary 3.5 [168])**.** *There is a deterministic distributed algorithm that for any radius $r \geq 1$, computes an r-neighborhood cover $\mathcal{N}$ within $\widetilde{O}(r)$ $\mathsf{CONGEST}$ rounds.*

Low-congestion cycle covers. The construction of FT cycle covers is based on the distributed construction of (c, d) cycle covers in the standard $\mathsf{CONGEST}$ model. In particular, we use the construction from [156, 155] that covers each edge $e = (u, v)$ by a cycle C_e such that $|C_e| = \widetilde{O}(\text{dist}_{G \setminus \{e\}}(u, v))$.

Fact 9 ([156, 155])**.** *There is a randomized algorithm* $\mathsf{ComputeCycCov}(G, D')$ *that for any n-vertex input graph $G = (V, E)$ and an input parameter D', computes, w.h.p., a cycle collection $\mathcal{C}$ with the following properties: (1) every edge $e \in E$ that lies on a cycle of length at most D' in G is covered by a cycle in $\mathcal{C}$ of length $\widehat{O}(D')$, and (2) each edge appears on $\widehat{O}(1)$ cycles. Alg.* $\mathsf{ComputeCycCov}(G, D')$ *runs in $\widehat{O}(D')$ rounds. In the output format, each vertex knows the edges of the cycles that cover each of its incident edges.*

Note that for Alg. $\mathsf{ComputeCycCov}$ does not require the graph G to be connected. This will be important in our context. This algorithm can also be made deterministic using the neighborhood covers of Theorem 26, see the full paper [88] for the proof of the following.

Observation 11. *The algorithm* $\mathsf{ComputeCycCov}(G_i, D')$ *of Fact 9 can be made deterministic using the neighborhood covering algorithm of Theorem 26. Additionally, in the output format of the algorithm, each vertex u knows a $\widehat{O}(1)$-bit unique identifier for each of the cycles it belongs to, as well as a full description of the cycle, obtained from both directions.*

Leader election. The broadcast algorithm of Theorem 12 (Chapter 5) implies a leader election algorithm. The proof of the following claim is given in the full paper [88].

Claim 31 (Byzantine Leader Election)**.** *Given a D–diameter, 3 edge-connected graph G and an adversarial edge e', assuming a linear upper bound $D' = cD$ on the diameter (for some constant $c \geq 1$), there exists a randomized algorithm* $\mathsf{AdvBroadcast}$ *that w.h.p elects a single leader known to all vertices in the graph within $\widetilde{O}(D^2)$ rounds.*

6.2 Compilers against a Single Adversarial Edge

We first describe the construction of $(1, e')$-FT cycle covers where e' is the adversarial edge in the graph. Then, we describe how to compile a single round using these cycles.

6.2.1 $(1, e')$-FT Cycle Cover

This section is mainly devoted to showing the following key lemma that computes a $(1, e')$-FT cycle cover, given a locally known covering family.

Lemma 20. *Given is a 3 edge-connected graph G, with a fixed unknown adversarial edge e'. Let L be an integer satisfying that for every edge $e = (u, v)$ it holds that* $\text{dist}_{G\setminus\{e,e'\}}(u, v) \leq L$*. Assuming that all vertices locally know a $(L, 1)$ covering family $\mathcal{G}$ of size ℓ, there exists a deterministic algorithm* ComputeOneFTCycCov *for computing a $(1, e')$ FT-cycle cover $\mathcal{C}$ with parameters $c = \widehat{O}(\ell), d = \widehat{O}(L)$ within $\widehat{O}(L^2 \cdot \ell)$ rounds.*

Since the computation of the $(L, 1)$ covering family is straightforward using known tools, we focus on proving Lemma 20. As a warm-up, we describe the construction assuming a reliable setting (with no adversarial edges). Then, we handle the real challenge of the $(1, e')$-FT cycle cover computation in the presence of an adversarial edge.

Warm-up: $(1, e')$-FT cycle covers in a reliable communication graph. The construction is based on applying the cycle cover algorithm of [156] on every subgraph G_i in the covering family $\mathcal{G}$. Specifically, given a locally known covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$, the algorithm proceeds in ℓ iterations. In each iteration i it applies the cycle cover algorithm ComputeCycCov(G_i, L) from Obs. 11 on the graph G_i with a diameter estimation L, resulting in a cycle collection $\mathcal{C}_i$. The final cycle collection is given by $\mathcal{C} = \bigcup_{i=1}^{\ell} \mathcal{C}_i$, that is, the union of all cycles computed in the ℓ iterations. We next analyze the construction.

Correctness. The round complexity, the cycle length, and the edge congestion bounds follow immediately by the construction. It remains to show that the cycle collection $\mathcal{C}$ is indeed a $(1, e')$-FT cycle cover. To see this, consider a fixed pair of edges $e = (u, v), e'$. We will show that $\mathcal{C}$ contains a cycle $C_{e,e'}$ that contains e and does not contain e'. An iteration i is defined to be *good* for the edge pair e, e' if

$$e' \notin G_i \ , \ e \in G_i \text{ and } \text{dist}_{G_i\setminus\{e\}}(u, v) \leq L \ .$$

Since, $\text{dist}_{G\setminus\{e,e'\}}(u, v) \leq L$, due to property (P1) of $\mathcal{G}$, there exists a good iteration i^* for every pair e, e'. We next show that e is successfully covered in iteration i^* by some cycle C_e.

By the properties of Alg. ComputeCycCov, in iteration i^* the edge e is covered by a cycle C of length $\widehat{O}(L)$. In addition, as $e' \notin G_{i^*}$ this cycle does not contain e' as required.

Alg. ComputeOneFTCycCov **(Proof of Lemma 20).** Given is a locally known covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$. The algorithm works in ℓ iterations, where in iteration i it performs the computation over the subgraph G_i. Since $\mathcal{G}$ is locally known, every vertex knows its incident edges in G_i, and ignores messages from other edges in that iteration. Every iteration i consists of two steps. In the first step, the vertices apply Alg. ComputeCycCov(G_i, L) of Obs. 11 over the graph G_i with diameter estimate L. This results in a cycle collection $\mathcal{C}'_i(u)$ for every vertex u. In the output format of Alg. ComputeCycCov, every cycle in $\mathcal{C}'_i(u)$ is presented by a tuple $(ID(C), C)$, where $ID(C)$ is the unique identifier of the cycle of size $\widehat{O}(1)$ bits, and C is the collection of the cycle edges[1]. Since e' might be in G_i, the cycles of $\mathcal{C}'_i(u)$ can be totally corrupted.

[1]Recall that in Alg. ComputeCycCov(G_i, L), each vertex receives the cycle description C from both di-

In the second step of iteration i, the vertices apply a verification procedure for their cycles in $\mathcal{C}'_i(u)$. Only verified cycles will then be added to the set of cycles $\mathcal{C}_i(u)$. In the analysis section, we show that for every reliable edge $e = (u, v) \neq e'$, there exists at least one cycle in $\mathcal{C}(u) = \bigcup_i^{\ell} \mathcal{C}_i(u)$ that covers e and does not contain e'. The third step of the algorithm handles the remaining adversarial edge, in case needed. We next elaborate on these steps in more detail. We focus on iteration i where the vertices communicate over the graph $G_i \in \mathcal{G}$.

Step (1) of iteration i: Cycle cover computation. The (fault-free) cycle cover algorithm ComputeCycCov of Obs. 11 is applied over the subgraph $G_i \in \mathcal{G}$, with parameter L. Since the graph G_i is locally known, each vertex can verify which of its incident edges lie on G_i and ignore the messages from the remaining edges. During the execution of ComputeCycCov(G_i, L), if a vertex u receives an illegal message, or different cycle descriptions with the same cycle ID, these messages are ignored, as well as future messages in that iteration. At the end of the execution of ComputeCycCov(G_i, L), each vertex u performs the following verification step on its output cycle set $\mathcal{C}'_i(u)$. The goal of this verification is to ensure each cycle in $\mathcal{C}'_i(u)$ corresponds to a legal cycle.

Step (2) of iteration i: Cycle verification. First, each vertex u performs a *local* inspection of its cycles in $\mathcal{C}'_i(u)$, and declares the iteration to be *faulty* if $\mathcal{C}'_i(u)$ contains at *least* one of the following:

1. A cycle of length $\widehat{\omega}(D)$;
2. An edge appearing in $\widehat{\omega}(1)$ cycles in $\mathcal{C}'_i(u)$;
3. A partial cycle (i.e., a walk rather than a cycle);
4. Inconsistency in a cycle description $(ID(C), C) \in \mathcal{C}'_i(u)$ as obtained through the two neighbors of u on C.

In the case where $\mathcal{C}'_i(u)$ is found to be faulty, u sets $\mathcal{C}_i(u) = \emptyset$, and will remain silent throughout this verification step. We will call such a vertex an *inactive* vertex. A vertex whose local inspection is successful is called *active*.

We now describe the global verification procedure for an active vertex u. The verification step is performed in *super-rounds* in the following manner. Each super-round consists of $c = \widehat{O}(1)$ rounds, which sets the upper bound on the number of cycles that an edge (u, v) participates in. A single super-round has sufficient bandwidth to exchange a single message through an edge (u, v) for each of the cycles on which (u, v) lies. We then explicitly enforce that in each super-round, each vertex u sends over an edge (u, v) at most *one* message per cycle $(ID(C), C) \in \mathcal{C}'_i(u)$ for which $(u, v) \in C$.

For a cycle $(ID(C), C) \in \mathcal{C}'_i(u)$, let v_C be the vertex with largest ID in the cycle description C obtained by u during Alg. ComputeCycCov(G_i, L). We note that the cycle description C is

rections, i.e., from its two neighbors on C. In case a vertex u obtained distinct cycle descriptions from its two neighbors, it omits the cycle from its cycle collection $\mathcal{C}'_i(u)$.

not necessarily correct, and in particular, it could be that $(ID(C), C) \notin \mathcal{C}'_i(v_C)$. For each cycle $(ID(C), C) \in \mathcal{C}'_i(u)$ such that $u = v_C$ (the cycle's leader), it initiates the following verification steps.

(2.1) A leader v_C of a cycle $(ID(C), C) \in \mathcal{C}'_i(v_C)$ sends the verification message $ver(C) = (ID(C), ID(v_C), ver)$ along its two incident edges on this cycle (i.e., in the clockwise and counter-clockwise directions).

(2.2) The verification messages are then propagated over the cycles for $R = \widehat{O}(L)$ super-rounds, where $\widehat{O}(L)$ is the upper bound on the maximal cycle length. Upon receiving a verification message $ver(C) = (ID(C), ID(v_C), ver)$, an active vertex u sends $ver(C)$ to a neighbor $w \in N(u)$ if the following conditions hold: (1) $(ID(C), C_u) \in \mathcal{C}'_i(u)$ for some cycle C_u, (2) v_C is the vertex with the highest ID in C_u, (3) w is a neighbor of u in C_u, and (4) u received the message $ver(C)$ from its second neighbor in the cycle C_u.

(2.3) A leader v_C of a cycle C such that $(ID(C), C) \in \mathcal{C}_i(v_C)$, which did not receive the verification message $ver(C)$ from both its neighbors on C within R super-rounds, initiates a *cancellation message*, $cancel(C) = (ID(C), ID(v_C), cancel)$, and sends it to both its neighbors in C. This indicates to the vertices on this cycle that the cycle should be omitted from their cycle collection.

(2.4) The cancellation messages $cancel(C)$ are propagated over the cycle C for R super-rounds in the following manner. Let τ_i be the first super-round of Step (2.3). In this super-round, v_C may start propagating the message $cancel(C)$ (if the conditions of (2.3) hold). Note, however, that the cancellation messages might also originate at the adversarial edge e'. Step (2.4) handles the latter scenario by augmenting the cancellation messages $cancel(C)$ with distance information. For every vertex u let d^1_u, d^2_u be the u-v_C distance on C along the first (second) u-v_C path in C. Note that u can locally compute d^1_u, d^2_u using the cycle description of C. A vertex u upon receiving a $cancel(C)$ message from its neighbor v on C acts as follows. Let d^j_u be the length of the v_C-u path on C that passed through v. Then, if the message $cancel(C)$ is received at u from v in super-round $r_j = \tau_i + d^j_u$, u *accepts* the cancellation message and sends it to its other neighbor on C. All other cancellation messages received by u in later or prior super-rounds are dropped.

(2.5) A leader v_C of a cycle $(ID(C), C) \in \mathcal{C}_i(v_C)$ that received the message $cancel(C)$ which it did not initiate from only *one* direction (i.e., from exactly one of its neighbors on C), broadcasts a cancellation message $cancel(i)$, i.e., canceling iteration i, to all the vertices in the graph by using the broadcast algorithm of Theorem 12 (Chapter 5) over the graph G. Since there is only one broadcast message $cancel(i)$ to be sent on that iteration, possibly by many cycle leaders, this can be done in the same time as a single broadcast operation (i.e., within $\widetilde{O}(D^2)$ rounds).

(2.6) A vertex that *accepts* a cancellation message $cancel(i)$ via the broadcast algorithm, omits all cycles obtained in this iteration i.

At the end of the i-th iteration, every vertex u defines a verified cycle set $\mathcal{C}_i(u)$. A cycle $(ID(C), C) \in \mathcal{C}'_i(u)$ is defined as *verified* by u if the following conditions hold (i) it received a verification message $ver(C)$ from both neighbors in C, (ii) any cancellation message $cancel(C)$ received by u has been dropped, and (iii) u did not accept a cancellation message $cancel(i)$ via the broadcast algorithm in Step (2.5). Every verified cycle $(ID(C), C) \in \mathcal{C}'_i(u)$ is added to the set $\mathcal{C}_i(u)$. Thus, $\mathcal{C}_i(u)$ consists of all verified cycles passing through u computed in iteration i. This concludes the description of the i-th iteration. The output of each vertex u is $\mathcal{C}(u) = \bigcup_{i=1}^{\ell} \mathcal{C}_i(u)$.

Step (3): Covering the adversarial edge. For a vertex u, an incident edge (u, v) is considered by u as *handled* if there exists a tuple $(ID(C), C) \in \mathcal{C}(u)$ such that $(u, v) \in C$. The goal of the third and final step is to cover the remaining unhandled edges. Every vertex u and an unhandled edge (u, v), broadcasts the edge (u, v) using the broadcast algorithm of Theorem 12. In the analysis section, we show that if there is an unhandled edge, then it must be the adversarial edge. The reason for broadcasting the edge (u, v) by its endpoints is to prevent the adversarial edge from initiating this step (despite the fact that all edges are covered). To cover (u, v), the endpoint with the larger identifier, say u, initiates a construction of a BFS tree T rooted at u in $G \setminus \{(u, v)\}$. Within $O(L)$ rounds, u and v learn the u-v tree path P. Then the cycle covering (u, v) is given by $C = (v, u) \circ P$. The cycle $(ID(C), C)$ is then[1] added to the cycle collection $\mathcal{C}(w)$ of every $w \in C$.

Correctness. We begin with showing that the tuples $\{\mathcal{C}(u)\}_{u \in V}$ computed in Step (2) induce legal cycles in G. That is, for every iteration i, a vertex $u \in V$, and every tuple $(ID(C), C) \in \mathcal{C}_i(u)$, we show that C is a cycle in G and $(ID(C), C) \in \mathcal{C}_i(w)$ for every $w \in C$. As we will see, the fact that the output of the algorithm induces (real) cycles will play a critical role in showing the adversarial edge is covered by a short cycle. We start with the following observation.

Observation 12. *Any maximal walk along which a message $ver(C') = (ID(C'), ID(v_{C'}), ver)$ is propagated towards some vertex w, either starts at $v_{C'}$ or at the adversarial edge e'. In addition, the walk must be a simple path.*

Proof. Since the message $ver(C')$ contains the identifier of the vertex $v_{C'}$, by Step (2.1) of the verification step, no other vertex but $v_{C'}$ initiates the verification message $ver(C')$. Hence, we can conclude that if the walk is not initiated by $v_{C'}$, it is initiated by the adversarial edge e'. Let P be the maximal walk along a message $ver(C')$ is propagated towards w. We next show P is a simple path. Assume by contraction there exists a vertex $v \in P$ with degree at least three in

[1]The ID of the cycle C can be obtained by appending the maximum ID vertex on C with a special tag indicating that the cycle is added in Step (3).

P. This contradicts Step (2.2), as for every vertex v, the number of neighbors in $N(v)$ which communicate the message $ver(C')$ with v is at most two. $\square$

Simple cycles. Recall that the adversarial edge is denoted by $e' = (v_1, v_2)$. A path P is denoted as *reliable* if it consists of only reliable edges (i.e., $e' \notin P$). For an $a \rightsquigarrow b$ directed path P, we denote the inverse path i.e., going from b to a, by $\overline{P}$. Fix an iteration i, and consider a vertex $u^* \in V$, and a tuple $(ID(C), C) \in \mathcal{C}_i(u^*)$. Since $(ID(C), C) \in \mathcal{C}_i(u^*)$, in Step (2) the vertex u^* received the verification message $ver(C) = (ID(C), ID(v_C), ver)$ from both neighbors on C denoted as w_1 and w_2, where v_C is the vertex with the highest ID in C. Let $P_1 = [u'_1, \ldots, u'_{k'} = u^*]$ and $P_2 = [u_1, \ldots, u_k = u^*]$ be the maximal directed walks on which the message $ver(C) = (ID(C), ID(v_C), ver)$ arrived to u^*. Let $\widehat{C} = P_1 \circ \overline{P}'_2$, where $P'_2 = P_2$ if $u_1 \neq v_1, v_2$, and $P'_2 = [u_2, \ldots, u_k]$ otherwise. That is, if both P_1 and P_2 start with the adversarial edge e', then e' in included in $\widehat{C}$ only once.

Our proof structure is as follows. We first show that $\widehat{C}$ is a simple cycle. Then we show that $\widehat{C} = C$ and that in addition, every vertex $w \in C$, includes $(ID(C), C)$ in the collection of its verified cycles $\mathcal{C}_i(w)$. Towards that goal, we begin with the following observation.

Observation 13. *Let $Q_1 = [a_1, a_2, \ldots, a_\ell = w]$ and $Q_2 = [b_1, b_2, \ldots, b_k = w]$ be two walks along which a message $ver(C') = (ID(C'), ID(v_{C'}), ver)$ is propagated. Then if $a_1 = b_1$ and $a_2 = b_2$, it holds that $Q_1 = Q_2$.*

Proof. We show that for every $i \geq 1$, $a_i = b_i$ by induction on i. For $i = 1, 2$, the claim follows from the assumption that $a_1 = b_1$ and $a_2 = b_2$. Assume that the claim holds for $i \geq 2$ and consider the vertices a_{i+1}, b_{i+1}. By the induction assumption, $a_i = b_i$. By the description of the walks Q_1 and Q_2, the vertex $w = a_i = b_i$ sent the message $ver(C)$ received from a_{i-1} to both a_{i+1} and b_{i+1}. By Step (2.2), w sends the message $ver(C)$ received from a_{i-1} to exactly one neighbor i.e., its second neighbor on the cycle C_w, where C_w is the unique cycle satisfying that $(ID(C), C_w) \in \mathcal{C}'_i(w)$. This implies that $a_{i+1} = b_{i+1}$. $\square$

Claim 32. *$\widehat{C}$ is a simple cycle.*

Proof. By the definition of P_1 and P_2, every vertex $w \in \widehat{C}$ sent the message $ver(C) = (ID(C), v_C, ver)$ towards u^*. Therefore, all vertices in P_1 and P_2 are active at the beginning of the verification step, and according to Step (2.2) for every $w \in \widehat{C}$, there exists a unique cycle, denoted as C_w, such that $(ID(C), C_w) \in \mathcal{C}'_i(w)$. We start with showing that $\widehat{C}$ is a cycle. By Obs. 12, we consider the following cases.

Case P_1 and P_2 start with v_C. In this case $\widehat{C}$ forms a cycle.

Case P_1 and P_2 start with e'. Without loss of generality, assume that P_1 starts with the (directed) edge (v_1, v_2). Assume toward contradiction that P_2 starts with (v_1, v_2) as well. By

Obs. 13, it then holds that $P_1 = P_2$ and $w_1 = w_2$, leading to a contradiction. Therefore P_1 and P_2 use e' in opposite directions, concluding that $\widehat{C}$ is a cycle.

Case P_1 starts with v_C and P_2 starts with the adversarial edge e'. We show that this case cannot occur. Assume without loss of generality that the path P_2 starts with the directed edge (v_1, v_2). We divide the analysis to two sub-cases.

Subcase $e' \notin P_1$. As P_1 is a reliable path and every vertex $w \in P_1$ sent the message $ver(C)$ towards u^*, it follows that the cycle description u^* received from its neighbor on P_1 (i.e, from $u'_{k'-1}$) during Alg. ComputeCycCov, is of the form $Q_1 \circ P_1$. Similarly, as all vertices on $P_2[v_2, u^*]$ are reliable, the cycle description u^* received from its neighbor on P_2 (i.e, from u_{k-1}), is of the form $Q_2 \circ P_2$. Since u^* is active at the beginning of the verification step, it follow that $C = Q_1 \circ P_1 = Q_2 \circ P_2$, and therefore $C = \overline{P_1} \circ P_3 \circ P_2$ for some simple v_C-v_1 path P_3. Moreover, since the path $\overline{P_1} \circ P_3$ is reliable, we can conclude that every $w \in \overline{P_1} \circ P_3$ received the path $\overline{P_1} \circ P_3$ as part of the cycle with identifier $ID(C)$, obtained by Alg. ComputeCycCov.

Next, since $(ID(C), C) \in \mathcal{C}_i(u)$, the vertex u^* did not receive a cancellation message $cancel(C)$ from v_C over the reliable path P_1. As P_1 is a reliable path, all vertices on P_1 holds the correct distance from v_C, and therefore we can conclude that v_C did not initiate a cancellation message. Hence, according to Step (2.3), v_C received the message $ver(C)$ from both its neighbors on the cycle - its neighbor in P_1 and its neighbor in $\overline{P_3}$.

Hence, according to Steps (2.1) and (2.2), the message $ver(C)$ propagated over the path P_3 towards v_1. As a result, during Step (2) the message $ver(C)$ is received by u^* over the concatenated path $\overline{P_3} \circ P_2$, in contradiction to the maximality of P_2. See Fig. 6.1(a) for an illustration.

Subcase $e' \in P_1$. First, assume that $(v_1, v_2) \in P_1$. Since P_2 starts with the same edge (v_1, v_2), by Obs. 13 it holds that $P_1[v_1, u] = P_2$. This contradicts the assumption that u^* received $ver(C)$ from two *different* neighbors over P_1 and P_2.

Next, assume that P_1 contains the directed edge (v_2, v_1). Let $w \in N(v_2)$ be the neighbor of v_2 such that w and v_1 are the neighbors of v_2 obtained by Alg. ComputeCycCov. As v_2 communicates the message $ver(C)$ only with its two neighbors in the cycle (i.e., v_1 and w), P_2 starts with the sub-path $[v_1, v_2, w] \subseteq P_2$, and P_1 contains the reveres sub-path $[w, v_2, v_1] \subseteq P_1$. As a result, by Obs. 13 it follows that $P_2 = \overline{P_1}[v_1, u^*]$. In particular, in holds that $u^* \in P_1[v_C, v_1]$, leading to a contradiction, as P_1 is a *simple* v_C-u^* path. See Fig. 6.1(b) for an illustration.

Finally, we show that $\widehat{C}$ is also simple. Assume toward contradiction that $\widehat{C}$ is not simple. Let v be the closest vertex to u^* on $\widehat{C}$ such that $v \in P_1 \cap P_2 \setminus \{v_C, u^*\}$. Note that because P_1 and P_2 are simple paths, and $P_1 \neq P_2$, if $\widehat{C}$ is not simple there exists such a vertex v. Since u^* received the message $ver(C)$ from two *different* neighbors w_1, w_2, it follows that $P_1[v, u] \neq P_2[v, u]$. As we choose v to be the closest vertex to u^* such that $v \in P_1 \cap P_2 \setminus \{v_C, u^*\}$, we conclude that v has two different neighbors in P_1 and P_2 denoted as a_1, a_2. Additionally, v

received the message $ver(C)$ from a different vertex a_3 over the path P_1. This contradicts Step (2.2), as the number of neighbors in $N(v)$ that send the message $ver(C)$ to v is at most two. $\square$

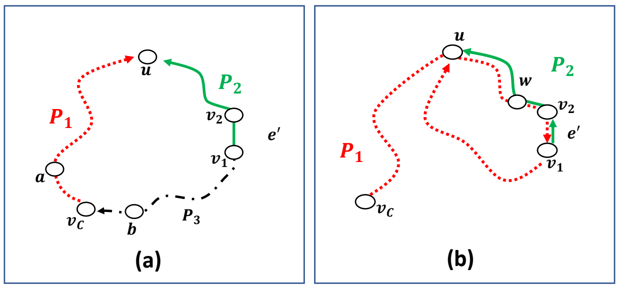

Figure 6.1: Proof of Claim 32: Illustrations of the case where P_1 (dotted red) starts with v_C, and P_2 (simple line green) starts with the adversarial edge (v_1, v_2). Fig. (a) illustrates the case where $e' \notin P_1$. Since u added the cycle to $\mathcal{C}_i(u)$, v_C did not send a cancellation message $cancel(C)$, and v_C received the message $ver(C)$ from an additional path P_3 (shown in dashed). In this case, the message $ver(C)$ propagated over the path $\overline{P_3} \circ P_2$ towards u, in contradiction to the maximality of P_2. Fig. (b) illustrates the case where $e' \in P_1$. As $P_1 \neq P_2$ the path P_1 contains the directed edge (v_2, v_1). Since v_2 send (or receive) the message $ver(C)$ only with its two neighbors v_1, w on the cycle C_{v_2}, it holds that $P_1[u, v_1] = \overline{P_2}$, in contradiction to the assumption that P_1 is a simple v_C-u path.

Claim 33. *At the end of the first step, it holds that $(ID(C), \widehat{C}) \in \mathcal{C}'_i(w)$ for every $w \in \widehat{C}$. In particular, it holds that $\widehat{C} = C$.*

Proof. By Claim 32, $\widehat{C}$ is a simple cycle. If $e' \notin \widehat{C}$ all vertices obtain the correct cycle description, and the claim follows. Next assume $e' \in \widehat{C}$, and let $w \in \widehat{C}$. Recall that in Alg. ComputeCycCov the cycle description is obtained in both directions. As all vertices in $\widehat{C}$ are active at the beginning of the verification step, w received the same cycle description from both its neighbors on $\widehat{C}$.

Denote $\widehat{C}$ by $\widehat{C} = C_1 \circ (v_1, v_2) \circ C_2$ where C_1 is a reliable w-v_1 path, and C_2 is a reliable v_2-w path. Hence, when obtaining the cycle description during Alg. ComputeCycCov(G_i, L), w received $Q_1 \circ (v_1, v_2) \circ C_1$ over C_1, and $Q_2 \circ (v_2, v_1) \circ C_2$ over C_2, for some paths Q_1, Q_2. Since w is active at the beginning of the verification step, w received *the same* cycle description in both directions, concluding that $Q_1 = C_2$ and $Q_2 = C_1$. It follows that $(ID(C), \widehat{C}) \in \mathcal{C}'_i(w)$.

In particular, for the vertex u^* it holds that $(ID(C), \widehat{C}) \in \mathcal{C}'_i(u^*)$. Since u^* is active at the beginning of the verification step, it obtained a single cycle with identifier $ID(C)$, and therefore $C = \widehat{C}$. $\square$

We now show that every vertex $w \in C$ adds the tuple $(ID(C), C)$ to its *verified* cycle collection $\mathcal{C}_i(w)$. Towards that goal, we start with the following auxiliary claim.

Claim 34. *Any cancellation message $cancel(C)$ received in Step (2.4) by some vertex $w \in C$ is dropped.*

Proof. Assume towards contradiction a vertex $w \in C$ received a $cancel(C)$ message during Step (2.4) that it did not drop. Consider the following cases:

- The message $cancel(C)$ received by w was initiated by v_C. In this case, we will show that also u^* (namely, the vertex that included $(ID(C), C)$ in its verified cycle set) received (and sent) the cancellation message as well, in contradiction to the assumption that the cycle is included in the output of u^*. In this case, the paths P_1 and P_2 start at v_C, that is, $P_1 = C[v_C, u^*]$ and $P_2 = C[u^*, v_C]$. Since C is simple, at least one of them, say P_2, does not contain the adversarial edge e'. According to Step (2.3), v_C sent the message $cancel(C)$ in both directions in the super-round τ_i. By Claim 33, all vertices in P_2 hold the correct cycle description of C, and the correct distances from v_C. As P_2 does not contain the adversarial edge, using an inductive argument it follows that every $v \in P_2$ at distance d^2_v from v_C on P_2, receives the message $cancel(C)$ in super-round $\tau_i + d^2_v$. According to Step (2.4), v also sends the message in super-round $\tau_i + d^2_v + 1$. It follows that u^* received the cancellation message $cancel(C)$ from the second direction on C (namely, along P_2), in super-round $\tau_i + d^2_{u^*}$, leading to a contradiction.

- The message $cancel(C)$ received by w was initiated by the adversarial edge $e' \in C$. We first show that v_C also received the message $cancel(C)$ by the end of Step (2.4). W.l.o.g assume that w received the message in direction 1 (clockwise). Since w did not drop the message, it follows that w received the message in super-round $\tau_i + d^1_w$, where d^1_w is the distance between v_C and w in direction 1. As all vertices in C hold the correct distances from v_C (Claim 33), using an inductive argument on the reliable w-v_C path $C[w, v_C]$, we conclude that v_C received the message $cancel(C)$ in super-round $\tau_i + |C| \leq \tau_i + R$ from direction 1.

  It remains to consider two cases.

  Case 1: v_C received the message $cancel(C)$ also from the second direction. In such a case, u^* also received the cancellation message, did not drop it, and sent it along the cycle. This contradicts the assumption that the cycle is included in the verified cycle set of u^*.

  Case 2: v_C received the cancellation message $cancel(C)$ only from a single direction (i.e., from one of its neighbors on C). According to Step (2.5), in this case v_C broadcasts a cancellation message $cancel(i)$ to all vertices. By the correctness of the broadcast algorithm, in Step (2.6), the vertex u^* cancels all its cycles defined in this iteration, leading to a contradiction.

$\square$

Claim 35. *For every $w \in C$ it holds that $(ID(C), C) \in \mathcal{C}_i(w)$.*

Proof. Recall that a vertex $w \in C$ adds a tuple $(ID(C), C)$ to its cycle collection $\mathcal{C}_i(w)$, only if (i) it received the verification message $ver(C)$ from both its neighbors on C, (ii) any cancellation message $cancel(C)$ received has been dropped, and (iii) it did not accept a cancellation message $cancel(i)$ via the broadcast algorithm in Step (2.5). We will show that all these conditions hold for w and C. Condition (ii) holds by Claim 34. As for condition (iii), as shown by Cor. 3 Chapter 5, no vertex accepts a false message initiated by the adversarial edge. Hence, if w accepts a message $cancel(i)$, it is initiated by some vertex in the graph. Since u^* did not accept such a message, by the correctness of the broadcast algorithm, w did not accept such a message as well. We are left to show condition (i) holds, and w received the verification message from both neighbors on C.

By Claim 33, it holds that $v_C \in C$. Let $P' = C[v_C, w]$ and $P'' = C[w, v_C]$ be the two v_C-w paths on C. Since C is simple, at least one of them say $P'' = C[w, v_C]$, does not contain the adversarial edge e'. By Step (2.1), at the beginning of Step (2), v_C sent $ver(C)$ to both neighbors on C. As a result, w received the message $ver(C)$ over the reliable path $\overline{P''}$. Additionally, as v_C did not send a cancellation message, v_C also received the message $ver(C)$ back over the path P''. Thus, it holds that w sent the message to its neighbor in P'', and therefore w also received the message from its other neighbor in P'. $\square$

Coverage. We begin by noting that in case the adversarial edge e' is not contained in a subgraph G_i, the cycles $\mathcal{C}'_i$ obtained by Alg. $\mathsf{ComputeCycCov}(G_i, L)$ are legal cycles in G, with valid cycle descriptions. That is, for every vertex u and a cycle identifier $ID(C)$ such that $(ID(C), C) \in \mathcal{C}'_i(u)$, for every $w \in C$ it holds that $(ID(C), C) \in \mathcal{C}'_i(w)$. Additionally, we observe that all cycles in $\mathcal{C}'_i$ pass the verification step, and therefore for every vertex $u \in V$ and a cycle $(ID(C), C) \in \mathcal{C}'_i(u)$, it holds that $(ID(C), C) \in \mathcal{C}_i(u)$.

Observation 14. *For a subgraph G_i such that $e' \notin G_i$, for every vertex $u \in V$ and a tuple $(ID(C), C) \in \mathcal{C}'_i(u)$, it holds that $(ID(C), C) \in \mathcal{C}_i(u)$.*

Proof. As $e' \notin G_i$, by the properties of Alg. $\mathsf{ComputeCycCov}$, the cycle collection $\mathcal{C}'_i$ admits the desired length and edge congestion. Therefore, all vertices in G are active at the beginning of the verification step. In consequence, by Step (2.1), the vertex with the highest ID in C denoted as v_C initiates the verification step and sends $ver(C)$ to both neighbors on C. Next, according to Step (2.2), all vertices on C send the verification messages in both directions. As a result, all vertices in C receive the verification message $ver(C)$ from both neighbors. As v_C also receives the message $ver(C)$ back from both neighbors on C, no cancellation message is sent and u adds the tuple $(ID(C), C)$ to its cycle collection $\mathcal{C}_i(u)$. $\square$

An edge $(u, v) \in E$ is said to be covered by the algorithm, if there exist a cycle C with ID $ID(C)$ such that $C \cap \{e, e'\} = \{e\}$, and $(ID(C), C) \in \mathcal{C}_i(w)$ for every $w \in C$. By the covering property of the graph family $\mathcal{G}$, we can now conclude that all reliable edges are covered.

Claim 36. *Every reliable edge $e \neq e'$ is covered by a cycle of length $\widehat{O}(L)$.*

Proof. By the promise of Lemma 20, for every reliable edge $e = (u, v)$, $\text{dist}_{G \setminus \{e,e'\}}(u, v) \leq L$. Hence, due to the covering property of $\mathcal{G}$, there exists a subgraph G_i containing all the edges of an L-length u-v path $P \subseteq G \setminus \{e, e'\}$, and in addition $G_i \cap \{e', e\} = \{e\}$. Thus, by the properties of $\mathsf{ComputeCycCov}(G_i, L)$, there exists a cycle $C \subseteq G_i$ of length $\widehat{O}(L)$, such that $(ID(C), C) \in \mathcal{C}'_i(u)$, and $e \in C$. Moreover, as $\mathsf{ComputeCycCov}(G_i, L)$ is executed correctly on G_i, for every $w \in C$ it holds that $(ID(C), C) \in \mathcal{C}'_i(w)$. By Obs. 14 it then follows that every vertex $w \in C$ adds the tuple $(ID(C), C)$ to $\mathcal{C}_i(w)$. Hence, e is covered by the end of the i-th iteration. □

Claim 37. *The adversarial edge $e' = (v_1, v_2)$ is covered by a cycle of length $\widehat{O}(L)$.*

Proof. First assume that at the beginning of Step (3) at least one of the vertices v_1 or v_2, say v_1 considered the edge (v_1, v_2) as handled. That is, at the beginning of Step (3) $(ID(C), C) \in \mathcal{C}(v_1)$ for some cycle C for which $(v_1, v_2) \in C$. By Claim 32, Claim 33 and Claim 35, it follows that e' is indeed covered.

Next, assume that at the beginning of Step (3) both v_1 and v_2 consider the edge e' as unhandled, and assume v_1 has a higher ID than v_2. According to Step (3), v_1 and v_2 broadcast the identifier of the edge e' using the broadcast algorithm of Theorem 12 (Chapter 5). By Claim 36 all reliable edges are covered during Step (2) of the algorithm. Hence, e' is the only uncovered edge, and the only message that is broadcast. By the properties of the broadcast algorithm, it must be initiated by a vertex and cannot be initiated by the adversarial edge. Therefore, all vertices in V accept a single message containing the identifier of e'.

Next, v_1 initiates a construction of a BFS tree T rooted at v_1 in $G \setminus \{e'\}$. Because we assumed that for every edge $e = (u, v)$ it holds that $\text{dist}_{G \setminus \{e\}}(u, v) \leq L$, the tree T is of depth $O(L)$, and contains a v_1-v_2 path P of size $O(L)$. Additionally, since all edges participating in the tree construction are reliable, the cycle $C = P \circ (v_1, v_2)$ covering e', is added to the cycle collection of all vertices in C. □

Congestion. For an iteration i, let $\mathcal{C}_i$ be the cycle collection obtained during the i-th iteration. We show the congestion of the cycles in $\mathcal{C} = \bigcup_i \mathcal{C}_i$ is at most $\widehat{O}(\ell)$.

Claim 38 (Congestion)**.** *For an iteration i each edge appears on $\widehat{O}(1)$ cycles in $\mathcal{C}_i$. Consequently, each edge appears on $\widehat{O}(\ell)$ cycles in $\mathcal{C} = \bigcup_{i=1}^{\ell} \mathcal{C}_i$.*

Proof. For an iteration i. First note that if $e' \notin G_i$, all edges participating in the i-th iteration are reliable. Hence, due to the properties of Alg. $\mathsf{ComputeCycCov}$, all cycles constructed during the execution of $\mathsf{ComputeCycCov}(G_i, L)$ are legal, and the congestion of each edge is $\widehat{O}(1)$. For an iteration i such that $e' \in G_i$, if an endpoint of an edge e detects high congestion in the cycles constructed by $\mathsf{ComputeCycCov}(G_i, L)$, it omits all cycles and becomes inactive. Hence, for each vertex u and an incident edge e, after the i-th iteration u outputs at most $\widehat{O}(1)$ cycles covering e. Because we have ℓ iterations, the total congestion of each edge is $\widehat{O}(\ell)$. □

We are now ready to prove Lemma 20.

Proof of Lemma 20. The covering property holds by Claim 36 and Claim 37. The congestion arguments hold by Claim 38. The length bound holds immediately, as all long cycles defined in bad iterations are omitted. It remains to bound the round complexity.

For each iteration i, performing Alg. $\mathsf{ComputeCycCov}(G_i, L)$ required $\widehat{O}(L)$ rounds. During the verification step, the congestion on each edge is kept bound by $\widehat{O}(1)$. In addition, since the length of each cycle is $\widehat{O}(L)$, steps (2.1)-(2.5) can be performed in $\widehat{O}(L)$ rounds. As for Step (2.6), in case some leaders broadcast a cancellation message $cancel(i)$ during iteration i, as all vertices broadcast the same message, we can view this step as performing a single execution of the broadcast algorithm of Theorem 12 in $\widetilde{O}(D^2) = \widetilde{O}(L^2)$ rounds.

Regarding the third step, the broadcast algorithm requires $\widetilde{O}(L^2)$ rounds. Constructing a BFS tree and performing the upcast and downcast steps requires $O(L)$ rounds. We conclude that the total round complexity of the algorithm is $\widehat{O}(L^2 \cdot \ell)$. □

We also show that if our graph G is not 3 edge-connected or with bounded diameter, our cycle cover algorithm has the guarantee to cover every reliable edge that lies on a reliable short cycle in G. That is, we achieve the following.

Corollary 7. *There exists a deterministic algorithm* $\mathsf{DetComputeOneFTCycCov}(G, L)$ *that given a graph G containing a single adversarial edge e' and a parameter L, returns a collection of cycles and paths $\mathcal{C}$ with the following property. Every reliable edge $(u, v) \neq e'$ for which* $\text{dist}_{G \setminus \{e', (u,v)\}}(u, v) \leq L$*, is covered by a reliable $\widehat{O}(L)$-length cycle $C \in \mathcal{C}$, such that $e' \notin C$ and $(u, v) \in C$.*

Proof of Theorem 22. By Obs. 6 setting $L = 7D$ satisfies the promise of Lemma 20. Hence, combining Lemma 20 with the covering family construction of Fact 6 with $L = 7D$, results in a deterministic $(1, e')$-FT cycle cover algorithm, with the following parameters.

Corollary 8. *There exists an r-round* deterministic *algorithm* $\mathsf{DetComputeOneFTCycCov}$ *for computing a $(1, e')$-FT cycle cover with parameters $d = \widehat{O}(D), c = \widehat{O}(D^2)$ and $r = \widehat{O}(D^4)$.*

As for the randomized algorithm, we begin with a construction of an $(L, 1)$ covering family with slightly weaker covering guarantees. The following Corollary follows by combining Theorem 19 and Theorem 12 of Chapter 5.

Corollary 9. *Given a 3 edge-connected graph G of diameter D with a fixed unknown adversarial edge e', one can compute in $\widetilde{O}(D^2)$ rounds a $(L = 7D, 1)$ covering family $\mathcal{G}$ of size $\ell = O(D \log n)$. The family $\mathcal{G}$ satisfies the following properties with high probability. For every $v, e, e' \in V \times V \times E \times E$, there exists a subgraph G_i such that (P1') $\mathrm{dist}_{G_i \setminus \{e,e'\}}(s, v) \leq L$ and (P2) $e \notin G_i = \emptyset$.*

To reduce the round complexity of Alg. ComputeOneFTCycCov, in Step (2.6) of the verification step, we use the nearly optimal randomized broadcast algorithms of Theorem 15 (Chapter 5) that work given that all vertices share a random seed of size $\widetilde{O}(1)$. To share this random seed, the algorithm begins with applying the randomized leader election algorithm of Claim 31 which takes $\widetilde{O}(D^2)$ rounds. Given a leader s, it shares a random seed r of $\widetilde{O}(1)$ bits by using the *deterministic* broadcast algorithm of Theorem 12 within $\widetilde{O}(D^2)$ rounds. From that point, the vertices have a shared seed and the future broadcast procedures will be based on that. We note that since the shared seed is only used to define the covering family, we can re-use the same shared seed in all future applications of the broadcast algorithm.

From now on, the randomized algorithm is exactly the same as Alg. ComputeOneFTCycCov, only that thanks to the shared seed, it uses the randomized broadcast algorithm of Theorem 15. This broadcast algorithm works in $\widetilde{O}(D)$ rounds. As it is applied ℓ many times, the round complexity is bounded by $\widehat{O}(\ell \cdot L + \ell \cdot D + L^2)$. The proof of Theorem 22 is completed by $\ell = \widetilde{O}(D)$ (see Cor. 9) and $L = 7D$.

6.2.2 General Compilers Given $(1, e')$-FT Cycle Cover

We next show that our $(1, e')$-FT cycle cover with parameters (c, d) yields a general compiler that translates any r-round distributed algorithm $\mathcal{A}$ into an equivalent algorithm $\mathcal{A}'$ that works in the presence e'.

Compiler against a single adversarial edge (Proof of Theorem 23). The compiler works in a round-by-round fashion, where every round of $\mathcal{A}$ is implemented in $\mathcal{A}'$ using a phase of $O(c \cdot d^2)$ rounds. At the end of the i-th phase, all vertices will be able to recover the original messages sent to them in round i of algorithm $\mathcal{A}$.

Compilation of round i. Let $\mathcal{C}$ be the cycle collection of the $(1, e')$-FT cycle cover. Fix a round i of Alg. $\mathcal{A}$ and let $M_{u \to v}$ be the message sent from u to v for every pair of neighbors $e = (u, v) \in E$ during the i-th round. In the i-th phase of $\mathcal{A}'$, the vertex u sends v the message $M_{u \to v}$ through e and *all* u-v paths $\mathcal{P}_{u,v} = \{C \setminus \{e\} \mid C \in \mathcal{C}, e \in C\}$. When sending the messages, each vertex on a path $P \in \mathcal{P}_{u,v}$ sends at most one message targeted from u to v. If a vertex w is requires to send at least two different messages from u to v, it omits both messages and sends a null message Φ over the cycle.

At the end of phase i, each vertex v sets the message $\widetilde{M}_{u,v}$ as its estimate for the message $M_{u\to v}$ sent by u in round i of $\mathcal{A}$. The estimate $\widetilde{M}_{u,v}$ is defined by applying the following protocol. In the case that v receives an identical message $M \neq \Phi$ from u through all the paths in $\mathcal{P}_{u,v}$, then $\widetilde{M}_{u,v} \leftarrow M$. Otherwise, $\widetilde{M}_{u,v} \leftarrow M'$ where M' is the message v received over the direct edge (u, v).

Correctness. We show that at the end of phase i for every edge $(u, v) \in E$ it holds that $\widetilde{M}_{u,v} = M_{u\to v}$. Consider the following two cases.
Case $e = e'$ is the adversarial edge. Since all u-v paths in $\mathcal{P}_{u,v}$ are reliable, all messages received by v over these paths must be identical. Thus, all the messages that v receives through the paths are identical, and equal to $M_{u\to v}$. By the definition of the $(1, e')$-FT cycle cover, $\mathcal{P}_{u,v} \neq \emptyset$. Hence, v accepts the correct message.
Case $e \neq e'$ is reliable. The message that u receives from v through the direct edge e is $M' = M_{u\to v}$. By the definition of the $(1, e')$-FT cycle cover, there exists a cycle $C \in \mathcal{C}$ covering e that does not contain e'. Hence, if all edges on C deliver the same message from u to v, it must be the message sent by u. Thus, if all messages v received through the paths $\mathcal{P}_{u,v}$ are identical and differ from Φ, they are equal to $M_{u\to v}$. Otherwise, v accepts the correct message $M' = M_{u\to v}$ delivered through the reliable edge (u, v).
Round complexity. Since each edge belongs to at most c cycles in the $(1, e')$-FT cycle cover $\mathcal{C}$, and as all cycles are of length at most d, the number of messages sent over an edge in a given phase is bounded by $c \cdot d$. Hence, each phase is implemented in $O(c \cdot d^2)$ rounds.

6.3 Compilers against Multiple Adversarial Edges

6.3.1 (f, F^*)-FT cycle cover*

At the heart of the compilers lies the construction of a (f, F^*)-FT cycle cover* in the adversarial CONGEST model that we describe in this section. Our main result is a deterministic construction of (f, F^*)-FT cycle covers* in the adversarial CONGEST model.

Lemma 21. *Given is an $(2f+1)$ edge-connected graph G with a fixed subset of unknown adversarial edges F^* of size f. Assuming all vertices locally know an $(L = 7fD, 2f)$ covering family $\mathcal{G}$ of size ℓ, there exists an r-round algorithm* ComputeFTCycCov *for computing an (f, F^*)-FT cycle cover* with parameters $d = \widehat{O}(L)$, $c = \widehat{O}(\ell \cdot L^2)$, and $r = \widehat{O}(\ell \cdot (fD \log n)^{O(f)})$ in the adversarial* CONGEST *model.*

The proof of Theorem 24 follows by combining Lemma 21 and Fact 6.
Our Approach. Before presenting the algorithm, we provide the high-level approach. Consider the following natural algorithm for computing an (f, F^*)-FT cycle cover*. Let $\mathcal{G}$ be an $(L, 2f)$ covering family for $L = O(fD)$. By applying the (fault-free) Alg. ComputeCycCov from Fact 9 on each subgraph $G_i \in \mathcal{G}$, we have the guarantee that all the reliable edges $E \setminus F^*$

are covered successfully as required by Definition 26. The key challenge is in determining whether the adversarial edges are covered as well. In particular, it might be the case that an edge $e \in F^*$ mistakenly deduces that it is covered, leading eventually to an illegal compilation of the messages sent through this edge. Note that unlike $(1, e')$-FT cycle covers, here an edge is covered only if it is covered by cycles of sufficiently large "flow".

Our approach is based on reducing the problem of computing an (f, F^*)-FT cycle cover* into the problem of computing $(1, e')$-FT cycle covers in *multiple* subgraphs for every $e' \in F^*$. Specifically, we define a covering family $\mathcal{G}$ with the following guarantee for each adversarial edge $e' \in F^*$: for every $F \subseteq G$, $|F| \leq f$, there exists a subgraph G_i containing a short cycle covering e' such that $G_i \cap (F^* \setminus \{e'\} \cup F) = \emptyset$. Since the covering guarantees for every $e' \in F^*$ are based on such "good" subgraphs G_i, it is safe to apply Alg. DetComputeOneFTCycCov (from Cor. 7) on these subgraphs (as they contain at most one adversarial edge). This approach also has a major caveat which has to do with the fact that the subgraph G_i is not necessarily 3 edge-connected and might not even be connected. In the single edge case, Alg. DetComputeOneFTCycCov is indeed applied on the input graph that is 3 edge-connected.

Recall that Alg. DetComputeOneFTCycCov is based on performing a verification step of the cycles, at the end of which we have the guarantee that at most one edge, corresponding to the adversarial edge, might not be covered. The third step of that algorithm then covers this edge, in case needed, using its fundamental cycle in the BFS tree. When applying Alg. DetComputeOneFTCycCov on the subgraph G_i, the situation is quite different. Since G_i is not necessarily connected, there might be potentially a large number of edges in G_i that are uncovered by cycles. Broadcasting the identities of these edges is too costly. For this reason, our algorithm applies the reduction in a more delicate manner.

Specifically, the algorithm applies Step (3) of Alg. DetComputeOneFTCycCov on the neighborhood cover of G_i (with a radius of $O(fD)$).

Alg. ComputeFTCycCov **(Proof of Lemma 21).**

Let $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ be a $(L, 2f)$ covering subgraph family that is locally known to all the vertices (from Definition 17). The algorithm iterates over the subgraphs in $\mathcal{G}$. In phase i, the algorithm considers the subgraph $G_i \in \mathcal{G}$ and applies two major steps. Let $E_i = \{e = (u, v) \in G_i \mid \text{dist}_{G_i \setminus \{e\}}(u, v) \leq L\}$ be the set of edges in G_i that are covered by a short cycle (of length at most $L + 1$) in G_i [1]. During the i-th phase, the goal is to cover the edges in E_i. The first step considers covering the reliable edges in $E_i \setminus F^*$, and the second step considers the adversarial edges $F^* \cap E_i$. Note that the endpoints of an edge e does not necessarily know if it belongs to E_i.

Step (1): Covering non-adversarial edges in G_i. The algorithm applies the deterministic $(1, e)$-FT cycle cover Alg. DetComputeOneFTCycCov(G_i, L') of Cor. 7 on the subgraph G_i with diameter estimate $L' = O(L \cdot \log^c n)$, where c is the constant of Definition 28 (in

[1]Note that the set E_i is unknown to the vertices in G.

the analysis part, it will be made clear why L' is set in this manner). When executing Alg. $\mathsf{ComputeOneFTCycCov}(G_i, L')$, Step (3) of that algorithm which covers the adversarial edge is omitted. In addition, in the verification step of Alg. $\mathsf{ComputeOneFTCycCov}(G_i, L')$ (Step 2.6), instead of using the broadcast algorithm of Theorem 12 against a single adversarial edge, we use the broadcast algorithm of Theorem 13 against f adversarial edges over the original graph G. If during the execution of Alg. $\mathsf{ComputeOneFTCycCov}(G_i, L')$, a vertex u receives an illegal message or that it needs to send too many messages through its incident edges (i.e., that exceeds the allowed $\widehat{O}(L'^2)$ congestion bound of Alg. $\mathsf{ComputeOneFTCycCov}(G_i, L')$), it cancels the i-th iteration in the following sense. It omits all its cycles computed in the i-th phase, and remains silent until the next phase.

For a vertex u, let $\mathcal{C}_i(u)$ be the cycle collection obtained by u during $\mathsf{ComputeOneFTCycCov}(G_i, L')$. Every vertex u that did not cancel the i-th phase, adds the cycles in $\mathcal{C}_i(u)$ to its final cycle collection $\mathcal{C}(u)$. Recall that the output of Alg. $\mathsf{ComputeOneFTCycCov}$ is given by a collection of tuples $\mathcal{C}_i(u) = \{(ID(C), C)\}$. At the end of Step (1), a vertex u considers its incident edge (u, v) as i-*handled* if there exists a tuple $(ID(C), C) \in \mathcal{C}_i(u)$ such that $(u, v) \in C$.

Step (2): Covering the adversarial edges in G_i. The goal of this step is to cover the adversarial edges of $E_i \cap F^*$. At the beginning of the step, the vertices locally compute an $(L, 1)$ covering family $\mathcal{G}_i = \{G_{i,1}, \ldots, G_{i,\ell_i}\}$ of size $\ell_i = \widetilde{O}(L^2)$ using Fact 6. The algorithm then proceeds in ℓ_i iterations, where in each iteration j the vertices perform the following sub-steps over the communication subgraph $G_{i,j} \in \mathcal{G}_i$.

(2.1) Compute an L neighborhood-cover $\mathcal{S}_{i,j} = \{S_{i,j,1}, \ldots, S_{i,j,k_{i,j}}\}$ by applying Theorem 26, and let $T_{i,j,q}$ be the spanning tree of each vertex-subset $S_{i,j,q}$.

(2.2) An edge (u, v) is *short bridgeless* if (i) (u, v) is not i-handled in Step (1), and (ii) there exists a tree $T_{i,j,q}$ containing u and v. For every short bridgeless edge e, the algorithm adds a cycle $C_e = \pi(u, v) \circ e$ to the cycle collection, where $\pi(u, v)$ is a u-v path in $T_{i,j,q}$. If during the execution of this step, a vertex u detects an incident edge with congestion above the limit, it omits all the cycles obtained in this step from its cycle collection $\mathcal{C}(u)$ and proceeds to the next sub-iteration.

Correctness. We first show the output collection admits the desire congestion and length bounds. The algorithm proceeds in ℓ phase. In each phase, all cycles obtained in Step (1) are of length $\widehat{O}(L)$, and the cycles obtained in Step (2) are of length $\widetilde{O}(L)$. Since every vertex u that detects an edge e with congestion above the limit omits the cycles computed in that iteration, each edge participates in at most $\widehat{O}(L^2)$ cycles. Hence, the total edge congestion is $\widehat{O}(\ell \cdot L^2)$.

Coverage. For an edge $e \in E$ and a subset $E' \subseteq E$ of size $|E'| \leq (f-1)$, the tuple (e, E') is *covered*, if there exists a cycle C with a cycle identifier $ID(C)$ satisfying $C \cap (F^* \cup E' \cup \{e\}) =$

$\{e\}$, and $(ID(C), C) \in \mathcal{C}(w)$ for every $w \in C$. An edge $e \in E$ is covered, if for every subset $E' \subseteq E$ of size $|E'| \leq (f-1)$, the tuple (e, E') is covered.

By the definition of an (f, F^*)-FT cycle cover*, we are left to show that all edges in E are covered. Given an edge $e = (u, v)$ and a subset $E' \subseteq E$ of size $|E'| \leq (f-1)$, a subgraph $G_i \in \mathcal{G}$ is *good* for the tuple (e, E') if (1) $(E' \cup F^* \cup \{e\}) \cap G_i = \{e\}$, and (2) $\text{dist}_{G_i \setminus \{e\}}(u, v) \leq L'$. We start with showing that for every tuple (e, E') there exists a good subgraph $G_i \in \mathcal{G}$.

Claim 39. *For every edge $e = (u, v)$ and a subset $E' \subseteq E$ of size $|E'| \leq (f-1)$, there exists a good subgraph $G_i \in \mathcal{G}$. Moreover, if a subgraph G_i is good for some tuple (e, E'), all cycles obtained during the execution of* ComputeOneFTCycCov(G_i, L') *in Step (1) are legal cycles in G with length $\widehat{O}(L)$ and edge congestion $\widehat{O}(L^2)$.*

Proof. By Obs. 6, for a $(2f+1)$ edge-connected graph G with diameter D and $L = 7fD$, for every edge $(u, v) = e \in E$, and every set $F \subseteq E$ of size $|F| \leq (f-1)$ it holds that $\text{dist}_{G \setminus \{F \cup F^* \cup \{e\}\}}(u, v) \leq L - 1$. Since $|E'| \leq (f-1)$, it also holds that $\text{dist}_{G \setminus (E' \cup F^* \cup \{e\})}(u, v) \leq L - 1$. Hence, as $|F^* \cup E'| \leq (2f-1)$, by the covering property of the $(2f, L)$ covering family $\mathcal{G}$, there exists a good subgraph G_i satisfying properties (1) and (2). Next, consider a subgraph $G_i \in \mathcal{G}$ that is good for some tuple (e, E'). Since $F^* \cap G_i \subseteq \{e\}$, the subgraph G_i contains at most one adversarial edge.

Recall that in Step (2.6) of the execution ComputeOneFTCycCov(G_i, L'), if a leader receives a cancellation message only from one direction, it broadcasts a cancellation message $cancel(i)$ using the broadcast algorithm of Theorem 13 (Chapter 5) against f adversarial edges, over the graph G. By Theorem 13 it follows that in such a case, all vertices in G will accept the message. The claim then follows from Theorem 22 and the properties of Alg. ComputeOneFTCycCov. □

We next show that all the reliable edges in G are covered. Towards that goal, we show that for an edge $e \notin F^*$ and a subset E', if a subgraph G_i is good for a tuple (e, E'), then (e, E') is covered by the end of the i-th phase.

Claim 40. *For an edge $e \notin F^*$ and a subset $E' \subseteq E$ of size $|E'| \leq (f-1)$, let $G_i \in \mathcal{G}$ be a good subgraph for (e, E'). Then after Step (1) of the i-th phase (e, E') is covered.*

Proof. Since G_i is good for (e, E'), $\text{dist}_{G_i \setminus \{e\}}(u, v) \leq L'$. Additionally, by Claim 39 the execution of ComputeOneFTCycCov(G_i, L') is performed correctly, and no vertex in V cancels the i-th phase. As e is reliable, by Cor. 7 we conclude that there exists a cycle C with cycle identifier $ID(C)$, such that $e \in C$, and $(ID(C), C) \in \mathcal{C}_i(w)$ for every $w \in C$. As no vertex cancels the i-th phase, w also adds the tuple $(ID(C), C)$ to its output set $\mathcal{C}(w)$. Finally, as $(E' \cup F^* \cup \{e\}) \cap G_i = \{e\}$ and $C \subseteq G_i$, then $(E' \cup F^* \cup \{e\}) \cap C = \{e\}$. The claim follows. □

Corollary 10 (Covering the Reliable Edges). *Every edge $e \notin F^*$ is covered.*

Proof. For an edge $e \notin F^*$, and a subset $E' \subseteq E$ of size $|E'| \leq (f-1)$, by Claim 39 there exists a subgraph $G_i \in \mathcal{G}$ that is good for (e, E'). Hence, by Claim 40 the tuple (e, E') is covered by the end of the i-th phase. $\square$

Claim 41 (Covering the Adversarial Edges). *Every* adversarial *edge $e = (u, v) \in F^*$ is covered.*

Proof. Fix a subset $E' \subseteq E$ of size $|E'| \leq (f-1)$. We will show that (e, E') is covered. By Claim 39 there exists a subgraph $G_i \in \mathcal{G}$ that is good for the tuple (e, E'). First assume that for at least one of the vertices u or v, say v, at the end of Step (1) the edge e is i-handled. By Claim 39 the execution of $\mathsf{ComputeOneFTCycCov}(G_i, L')$ is performed correctly, and therefore (e, E') is covered by the end of Step (1) of the i-th phase.

Next, assume that at the end of Step (1), both v and u considers the edge e as not i-handled. Let $G_{i,j} \in \mathcal{G}_i$ be a subgraph of G_i satisfying that (P1) $\text{dist}_{G_{i,j}}(u, v) \leq L$, and (P2) $e \notin G_{i,j}$. By Fact 6, there exists such a subgraph $G_{i,j}$. Since $e \notin G_{i,j}$ the subgraph $G_{i,j}$ contains no adversarial edges. Therefore, the computation of Step (2.1) is performed successfully, resulting in an L-neighborhood-cover $\mathcal{S}_{i,j}$. Since $\text{dist}_{G_{i,j} \setminus \{e\}}(u, v) \leq L$, there exists a cluster $S_{i,j,k} \in \mathcal{S}_{i,j}$ such that $u, v \in S_{i,j,q}$. Hence, the edge $e = (u, v)$ is a short bridgeless edge. Therefore, according to Step (2.2), a cycle C_e with identifier $ID(C_e)$ of length $\widetilde{O}(L)$ such that $(u, v) \in C_e$ is added to the cycle collection. That is, every vertex $w \in C_e$ adds the tuple $(ID(C_e), C_e)$ to its output set $\mathcal{C}(w)$.

We next show that during the j-th iteration of Step (2), the edge (u, v) is the only short bridgeless edge. It will then imply that during iteration j, a single cycle is added to the cycle collection, and therefore the iteration is not canceled due to large congestion by any of the vertices in V. For every reliable edge $(w_1, w_2) \in G_i$, if there exists a tree $T_{i,j,k}$ containing both w_1 and w_2, by the definition of neighborhood covers it holds that $\text{dist}_{G_{i,j} \setminus \{(w_1, w_2)\}}(w_1, w_2) = O(L \cdot \log^c n) \leq L'$. Since $e \notin G_{i,j}$, it implies that $\text{dist}_{G_i \setminus \{e, (w_1, w_2)\}}(w_1, w_2) \leq L'$. Hence, by Claim 40 and Cor. 7, the edge (w_1, w_2) is covered in Step (1) of the i-th phase and therefore both w_1 and w_2 consider the edge as i-handled. Thus, (w_1, w_2) is not a short bridgeless edge. It follows that no vertex cancels the j-th iteration in Step (2.2). Additionally, as $C_e \subseteq G_i$, it holds that $C_e \cap (E' \cup F^*) = \{e\}$, and therefore the tuple (e, E') is covered. $\square$

Finally, for the sake of the implementation of the general compilers in the next section, we note that all the cycles and the paths in the (f, F^*)-FT cycle cover* are simple.

Observation 15. *All cycles and paths in an (f, F^*)-FT cycle cover* are simple.*

Proof. In every phase i, by the properties of Alg. $\mathsf{ComputeOneFTCycCov}(G_i, L')$, all cycles and paths obtained in Step (1) are simple. Additional, in Step (2), for every short bridgeless edge e, for which the algorithm adds a cycle $C_e = \pi(u, v) \circ e$ to the cycle collection, every vertex $w \in C_e$ adds a single tuple $(ID(C_e), C_e)$ to its final output set $\mathcal{C}(w)$. Hence the degree of every vertex in the cycle (or truncated cycle) obtained in Step (2) is at most two and therefore the cycle (or path) is simple. $\square$

Round complexity. The algorithm proceeds in ℓ phases. Consider phase i. Applying Alg. $\mathsf{DetComputeOneFTCycCov}(G_i, L')$ in Step (1) takes $\widehat{O}(L^4)$ rounds. Applying the broadcast alg. of Theorem 13 used during the verification step (Step (2.6)) takes $O(fD\log n)^{O(f)}$ rounds. In Step (2), the algorithm iterates over $\ell_i = \widetilde{O}(L^2)$ subgraphs $\mathcal{G}_i$. In each iteration, applying the L-neighborhood cover algorithm of Theorem 26 takes $\widetilde{O}(L)$ rounds. The total round complexity is bounded by $\widehat{O}(\ell \cdot (fD\log n)^{O(f)})$. Note that one can also reduce the edge congestion by using the randomized algorithm for $(1, e')$-FT cycle cover of Theorem 22.

6.3.2 General Compilers Given (f, F^*)-FT cycle cover*

We next show that our (f, F^*)-FT cycle cover* yields a general compiler that translates any r-round distributed algorithm $\mathcal{A}$ into an equivalent algorithm $\mathcal{A}'$, that works in the presence of f adversarial edges with round complexity $r' = O(r \cdot c \cdot d^3)$. Throughout, we use the following definition for a minimum s-v cut defined over a collection of s-v *paths*.

Definition 29 (Minimum (Edge) Cut of a Path Collection)**.** *Given a collection of s-v paths $\mathcal{P}$, the minimum s-v cut of $\mathcal{P}$, denoted as MinCut$(s, v, \mathcal{P})$, is the minimal number of edges appearing on all the paths in $\mathcal{P}$. I.e., MinCut$(s, v, \mathcal{P}) = x$ implies that there exists a collection of x edges E' such that for every path $P \in \mathcal{P}$, it holds that $E' \cap P \neq \emptyset$.*

General compiler given an (f, F^*)-FT cycle cover* (Proof of Theorem 25). The compiler works in a round-by-round fashion, where every round of $\mathcal{A}$ is implemented within a phase of $O(c \cdot d^3)$ rounds.

Compilation of round i. Let $\mathcal{C}$ be the given (f, F^*)-FT cycle cover*. Fix a round i in $\mathcal{A}$, and an edge $e = (u, v) \in E$, and let $M_{u \to v}$ be the message sent from u to v in round i. Let $\mathcal{P}_{u,v} = \{C \setminus \{e\} \mid C \in \mathcal{C}, e \in C\}$ be the collection of u-v paths obtained from the cycles covering e in $\mathcal{C}$. In the i-th phase of Alg. $\mathcal{A}'$, the vertex u sends v the message $M_{u \to v}$ through the direct edge e, as well as through *all* the u-v paths in $\mathcal{P}_{u,v}$. The message $M_{u \to v}$ is augmented with the cycles ID and the path description, where every vertex in the path augments the message with its own ID [1]. On every path (i.e., for every cycle ID), each vertex sends at most one such message.

At the end of phase i, every vertex v computes the message $\widetilde{M_{u,v}}$ as its estimate for the message $M_{u \to v}$ for every $u \in N(v)$. Let M' be the message v received over the directed edge (u, v). For a message M received by v, let $\mathcal{P}_M$ be the path collection on which v received this message during the i-th phase, such that $(u, v) \notin P$ for every path $P \in \mathcal{P}_M$. If

[1]We note that in the compilers against a single edge, it was not needed to send the path information.

MinCut$(u, v, \mathcal{P}_{M'}) \geq f$, then v sets $\widetilde{M}_{u,v} = M'$. Otherwise, v sets $\widetilde{M}_{u,v} = M$, for some message M satisfying MinCut$(u, v, \mathcal{P}_M) \geq f$, breaking ties arbitrarily.

Correctness. Fix a phase i and an edge $e = (u, v) \in E$. We show that at the end of the i-th phase, it holds that $\widetilde{M}_{u,v} = M_{u \to v}$. We first show that MinCut$(u, v, \mathcal{P}_{M_{u\to v}}) \geq f$.

Claim 42. *At the end of the i-th phase it holds that MinCut$(u, v, \mathcal{P}_{M_{u\to v}}) \geq f$.*

Proof. By the definition of the (f, F^*)-FT cycle cover* $\mathcal{C}$, for every subset $E' \subseteq E$ of size $|E'| \leq f - 1$, there exists a cycle $C \in \mathcal{C}$ such that $C \cap (E' \cup F^* \cup \{e\}) = \{e\}$. It then follows that for the set $\mathcal{P} = \{P \in \mathcal{P}_{u,v} \mid P \cap F^* = \emptyset\}$, it holds that MinCut$(u, v, \mathcal{P}) \geq f$. Since $P \cap F^* = \emptyset$ for every path $P \in \mathcal{P}$, for the correct message $M_{u\to v}$ it holds that $\mathcal{P} \subseteq \mathcal{P}_{M_{u\to v}}$ and therefore MinCut$(u, v, \mathcal{P}_{M_{u\to v}}) \geq f$. □

If the edge (u, v) is reliable, then $M' = M_{u\to v}$, and due to Claim 42 it also holds that MinCut$(u, v, \mathcal{P}_{M_{u\to v}}) \geq f$. Therefore v sets $\widetilde{M}_{u,v} = M' = M_{u\to v}$. Next, assume $(u, v) \in F^*$ is adversarial. We claim that for any message $M \neq M_{u\to v}$, it holds that MinCut$(u, v, \mathcal{P}_M) \leq f - 1$. For a path $P \in \mathcal{P}_M$, since $M \neq M_{u\to v}$, the message M which propagated over the path P, was initiated by some adversarial edge in F^*. We claim that the path description of P contains an edge $e' \in F^*$, as for the last adversarial edge in P, the vertex v will receive its ID as part of the path description. Additionally, since the edge (u, v) does not appear in any path in $\mathcal{P}_M$, it follows that $P \cap (F^* \setminus \{e\}) \neq \emptyset$ for every path $P \in \mathcal{P}_M$. Hence, MinCut$(u, v, \mathcal{P}_M) \leq |F^* \setminus \{e\}| = f - 1$. We can then conclude that $M_{u\to v}$ is the only message for which MinCut$(u, v, P_M) \geq f$ and therefore $\widetilde{M}_{u,v} = M_{u\to v}$ as desired.

Round complexity. All cycles (and paths) in the (f, F^*)-FT cycle cover* have length at most d, and edge congestion at most c. Since each edge belongs to c cycles of length d, the number of messages sent over an edge in a given phase is bounded by $c \cdot d$. Since each message sent over a path P is augmented with the path description of P, the messages have size $O(d \log n)$. Hence, in every phase, each edge needs to send $c \cdot d$ messages, of size $O(d \log n)$ along paths of size d. Therefore each phase requires $O(c \cdot d^3)$ rounds. As a result, an r-round algorithm $\mathcal{A}$ is simulated in $\mathcal{A}'$ by phase of $O(r \cdot c \cdot d^3)$ rounds.

6.3.3 Lower Bounds (Proof of Theorem 21)

In this section, we present a lower bound for the quality of the FT-cycle covers with parameters c, d, where the quality is measured by the summation $c + d$ (congestion + dilation). At the heart of our lower bound argument lies a graph-theoretical characterization of the tradeoff between congestion and dilation in a collection of s-v paths with a sufficiently large flow. Given an (unweighted) graph $G = (V, E)$, and a pair of vertices s, v, a *route set* is a collection of s-v paths in G of sufficiently large flow. Our goal is then to characterize the tension between the length of these paths and their overlap subject to some threshold on the flow of these paths.

Congestion vs. dilation tradeoff in route-sets. Given a graph $G = (V, E)$, a pair of vertices s, v and a route-set $\mathcal{P}$ of s-v paths, define the *dilation* of $\mathcal{P}$ by $\text{dilation}(\mathcal{P}) = \max_{P \in \mathcal{P}} |P|$, and the congestion of $\mathcal{P}$ by $\text{congestion}(\mathcal{P}) = \max_{e \in E} |\{P \in \mathcal{P} \mid e \in P\}|$. The next lemma shows that if the flow of the route-set is required to be sufficiently large, i.e., at least t, then there are graphs in which the congestion + dilation of these paths *must* be exponentially large in t.

Lemma 22. *For any parameters $t \geq 1$, $\rho \geq 1$ and $D = \Omega(t)$, there exists a $(t+1)$ edge-connected graph $G = (V, E)$ with diameter D, and two vertices $u, v \in V$ with the following property. For* any *collection of u-v paths $\mathcal{P}$ such that MinCut$(u, v, \mathcal{P}) \geq t+1$ it holds that if* $\text{dilation}(\mathcal{P}) = \rho$ *then* $\text{congestion}(\mathcal{P}) = \Omega((\frac{D-1}{t+1})^{t+1}/(t \cdot \rho))$.

The Graph construction. We start with describing the lower bound graph denoted as G^*. The graph G^* consists of $t+1$ layers of edges $E_0, E_1, \ldots, E_t$ described described in an inductive manner. Let $\widehat{D} = \frac{D-1}{t+1}$.

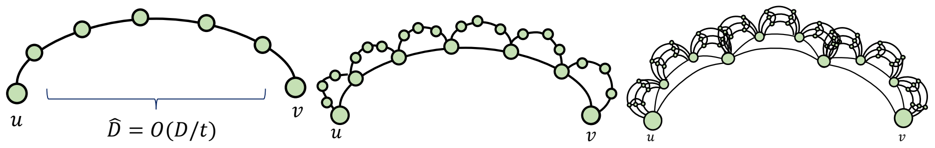

Figure 6.2: An illustration of the graph construction for $t = 2$. Left: The first layer consists of a simple u-v path of length $\widehat{D}$. Middle: Illustration of the first two layers E_0 and E_1. Every layer consists of $\widehat{D}$ lengths paths covering the edges of the previous layer. Right: The constructed graph. The last layer E_t covers each of the edges in E_{t-1} with $t+1$ edge disjoint paths that are connected via cliques.

Layer 0**.** We introduce two designated vertices $u, v \in V$. The vertices u and v are connected via a path of length $\widehat{D}+1$, denoted as $P_0 = (u, u_1, \ldots u_{\widehat{D}}, v)$. See Fig. 6.2 (Left) for an illustration of the first layer E_0.

Layer i **for** $1 \leq i \leq t-1$**.** The i-th layer E_i consists of $\widehat{D}$-length paths covering the edges in E_{i-1}. For every edge $e = (w_1, w_2) \in E_{i-1}$, we connect w_1 and w_2 by a path $P_e = (w_1, u_{e,1} \ldots u_{e,\widehat{D}-1}, w_2)$ of length $\widehat{D}$. We then define $E_i = \bigcup_{e \in E_{i-1}} E(P_e)$. See Fig. 6.2 (Middle) for an illustration of the construction after adding the first layer of edges E_1 to the basic construction.

Layer t**.** The last layer E_t covers each of the edges in E_{t-1} with $t+1$ edge disjoint paths that are connected connected via cliques. For every edge $e = (w_1, w_2) \in E_{t-1}$, we introduce $t+1$ edge-disjoint w_1-w_2 paths of length $\widehat{D}$ denoted as $\mathcal{P}_e = \{(w_1, u^i_{e,1}, \ldots, u^i_{e,\widehat{D}}, w_2)\}_{i=1}^t$. Additionally, each set of vertices $u^1_{e,j}, \ldots u^{k-1}_{e,j}$ for $j \in [1, \widehat{D}-1]$ form a clique. This completes the description of the lower bound graph, as illustrated in Fig. 6.2 (Right).
Correctness. We start with showing the constructed graph G^* is indeed $(t+1)$ edge-connected, with diameter at most D.

Observation 16. *The graph G^* is $(t+1)$ edge-connected and with diameter at most D.*

Proof. For every two vertices $w_1, w_2 \in V$ by te definition of the last layer E_t, there exists $t+1$-edge disjoint w_1-w_2 paths in E_t. Next, we bound the diameter of G^*. Consider two vertices $w_1, w_2 \in V$. By the definition of the graph G^*, the distance between w_1 and w_2 from the path P_0 is at most $t\widehat{D}/2$. Since the length of P_0 is $\widehat{D}+1$, by the triangle inequality it holds that:

$$\text{dist}_{G^*}(w_1, w_2) \leq t\widehat{D}/2 + t\widehat{D}/2 + \widehat{D} + 1 = (t+1)\widehat{D} + 1 = D$$

□

The next claim completes the proof of Lemma 22.

Claim 43. *For any u-v route-set $\mathcal{P}$ with MinCut$(u, v, \mathcal{P}) \geq t+1$ and* $\text{dilation}(\mathcal{P}) = \rho$*, it holds that* $\text{congestion}(\mathcal{P}) = \Omega((\frac{D-1}{t+1})^{t+1}/(\rho \cdot t))$.

Proof. Let $\mathcal{P}$ be a collection of u-v paths such that $\text{MinCut}(u, v, \mathcal{P}) \geq t+1$ with $\text{dilation}(\mathcal{P}) = \rho$. Consider a collection of t–tuples $\mathcal{F} \subseteq E_0 \times E_1 \cdots \times E_t$, defined as follows. A tuple $(e_0, \ldots e_{t-1})$ is in $\mathcal{F}$ if for every $i \in [0, t-1]$ it holds that (i) $e_i \in E_i$, and (ii) e_i is on the unique path covering e_{i-1}, added in the i-th layer. Note that by the definition of the graph G^*, the size of $\mathcal{F}$ is $\widehat{D}^t$. In order to bound the congestion of $\mathcal{P}$ we begin with bounding the size of the route set $\mathcal{P}$ using the cardinality of $\mathcal{F}$.

For a u-v path $P' \in \mathcal{P}$, let $S(P') = \{F \in \mathcal{F} \mid P' \cap F = \emptyset\}$ be the tuples in $\mathcal{F}$ which do not intersect with P'. We next show that $|P'| = \Omega(\widehat{D} \cdot |S(P')|)$. For every $F = (e_0, \ldots, e_{t-1}) \in S(P')$, because P' is a u-v path and $P' \cap F = \emptyset$, P' contains a sub-path P_F between the two endpoints of the edge e_{t-1}, such that $P_F \subseteq E_t$. By the definition of E_t, it holds that $|P_F| \geq \widehat{D}$. Additionally, by the definition of $\mathcal{F}$, for every two *different* tuples $F_1 = (e_0, \ldots, e_{t-1})$ and $F_2 = (e'_0, \ldots, e'_{t-1})$ in $S(P')$, it holds that $e_{t-1} \neq e'_{t-1}$, and therefore $P_{F_1} \cap P_{F_2} = \emptyset$. Hence, for every tuple $F \in S(P')$ we can account P' with $\widehat{D}$ unique edges. As a result, $|P'| = \Omega(\widehat{D} \cdot |S(P')|)$.

Next, as $\text{MinCut}(u, v, \mathcal{P}) \geq t+1$, for every tuple $F \in \mathcal{F}$ of t edges, there exists a path $P' \in \mathcal{P}$ such that $F \cap P' = \emptyset$ (i,e,. $F \in S(P')$). Since the length of each path $P' \in \mathcal{P}$ is at most ρ, it holds that $|S(P')| \leq \rho/\widehat{D}$.

As $|\mathcal{F}| = \widehat{D}^t$, we conclude that the collection $\mathcal{P}$ contains at least $\widehat{D}^t \cdot \widehat{D}/\rho$ different paths. Because the degree of the vertex v is $2f+1$, there exists an edge (w, v) that participates in at least $\widehat{D}^{t+1}/(\rho \cdot (2f+1))$ paths in $\mathcal{P}$. Hence,

$$\text{congestion}(\mathcal{P}) \geq \widehat{D}^{t+1}/(\rho \cdot (2f+1)) = \left(\frac{D-1}{t+1}\right)^{t+1} /(\rho \cdot (2f+1)) \,.$$

□

From Lemma 22 it follows that there exists a graph G^* with diameter D such that any collection of $t+1$ disjoint paths (i.e., with $\text{congestion}(\mathcal{P}) = 1$), must contain a long path.

Corollary 11. *For any parameters $t \geq 1$, and $D = \Omega(t)$, there exists a $(t+1)$ edge-connected graph $G = (V, E)$ with diameter D, and two vertices $u, v \in V$ such that* any *collection of u-v* disjoint *paths $\mathcal{P}$ of size $|\mathcal{P}| = t+1$, contains a path $P \in \mathcal{P}$ of length $|P| = \Omega((\frac{D-1}{t+1})^{t+1}/t)$.*

Finally, we prove Theorem 21 and show a lower bound on the quality of f-FT cycle covers using Lemma 22. **Proof of Theorem 21**

Proof. Let $G^* = (V, E)$ be the n-vertex $(f+1)$ edge-connected graph with diameter D, with two designated vertices $u, v \in V$ of Lemma 22. Consider the graph $G' = G^* \cup \{(u, v)\}$, obtained by adding the edge (u, v) to the graph G^*. Let $\mathcal{C}$ be an f-FT cycle cover for G' with parameters c, d, and let $\mathcal{P} = \{C \setminus \{(u, v)\} \mid C \in \mathcal{C}, \text{ and } (u, v) \in C\}$ be the collection of u-v paths induced by the cycles in $\mathcal{C}$. By the definition of the f-FT cycle cover $\mathcal{C}$, for every subset $E' \subseteq E$ of size $|E'| \leq f-1$, there exists a cycle $C \in \mathcal{C}$ such that $C \cap (E' \cup \{(u, v)\}) = \{(u, v)\}$. It then follows that MinCut$(u, v, \mathcal{P}) \geq f$. As $(u, v) \notin P$ for every $P \in \mathcal{P}$, the set $\mathcal{P}$ is also a u-v rout-set in G^*. Hence, by Lemma 22 it holds that dilation$(\mathcal{P})$ + congestion$(\mathcal{P}) = (\frac{D}{f})^{\Omega(f)}$, and therefore $c + d = (\frac{D}{f})^{\Omega(f)}$. $\square$

Bibliography

[1] Ittai Abraham, T.-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 317–326, 2019.

[2] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected O(1) rounds, expected o(n^2) communication, and optimal resilience. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, pages 320–334, 2019.

[3] Edgar D Adrian. The impulses produced by sensory nerve endings. *The Journal of physiology*, 61(1):49–72, 1926.

[4] Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 32–50. Springer, 2011.

[5] Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A biological solution to a fundamental distributed computing problem. *science*, 331(6014):183–185, 2011.

[6] James B. Aimone, Yang Ho, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Yipu Wang. Provable advantages for graph algorithms in spiking neural networks. In Kunal Agrawal and Yossi Azar, editors, *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 35–47. ACM, 2021.

[7] James B. Aimone, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Helen Xu. Dynamic programming with spiking neural computing. In Thomas E. Potok and

Catherine D. Schuman, editors, *Proceedings of the International Conference on Neuromorphic Systems, ICONS 2019, Knoxville, Tennessee, USA, July 23-25, 2019*, pages 20:1–20:9. ACM, 2019.

[8] Melissa J Allman, Sundeep Teki, Timothy D Griffiths, and Warren H Meck. Properties of the internal clock: first-and second-order principles of subjective time. *Annual review of psychology*, 65:743–771, 2014.

[9] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences*, 58(1):137–147, 1999.

[10] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

[11] Douglas B Armstrong, Arthur D Friedman, and Premachandran R Menon. Design of asynchronous circuits assuming unbounded gate delays. *IEEE Transactions on Computers*, 100(12):1110–1120, 1969.

[12] John Augustine, Gopal Pandurangan, and Peter Robinson. Fast byzantine leader election in dynamic networks. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 276–291, 2015.

[13] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Fast distributed network decompositions and covers. *J. Parallel Distributed Comput.*, 39(2):105–114, 1996.

[14] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990.

[15] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 514–522, 1990.

[16] Anindo Bagchi and S. Louis Hakimi. Information dissemination in distributed systems with faulty units. *IEEE Transactions on Computers*, 43(6):698–710, 1994.

[17] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002, Cambridge, MA, USA, September 13-15, 2002, Proceedings*, pages 1–10, 2002.

[18] Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO*

2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings, volume 3621 of *Lecture Notes in Computer Science*, pages 361–377. Springer, 2005.

[19] Amos Beimel and Lior Malka. Efficient reliable communication over partially authenticated networks. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*, pages 233–242. ACM, 2003.

[20] Mihir Bellare and John Rompel. Randomness-efficient oblivious sampling. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 276–287. IEEE, 1994.

[21] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1988.

[22] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla, and Kwabena Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[23] Piotr Berman, Krzysztof Diks, and Andrzej Pelc. Reliable broadcasting in logarithmic time with byzantine link failures. *Journal of Algorithms*, 22(2):199–211, 1997.

[24] Piotr Berman and Juan A. Garay. Cloture votes: n/4-resilient distributed consensus in t+1 rounds. *Math. Syst. Theory*, 26(1):3–19, 1993.

[25] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 410–415. IEEE Computer Society, 1989.

[26] Martin Biely and Ulrich Schmid. Message-efficient consensus in presence of hybrid node and link faults. 2001.

[27] Martin Biely, Ulrich Schmid, and Bettina Weiss. Synchronous consensus under hybrid process and link failures. *Theoretical Computer Science*, 412(40):5602–5630, 2011.

[28] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[29] Jaroslaw Blasiok. Optimal streaming and tracking distinct elements with high probability. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete*

Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018, pages 2432–2448, 2018.

[30] Douglas M Blough and Andrzej Pelc. Optimal communication in networks with randomly distributed byzantine faults. *Networks*, 23(8):691–701, 1993.

[31] Greg Bodwin, Michael Dinitz, and Caleb Robelle. Optimal vertex fault-tolerant spanners in polynomial time. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2924–2938. SIAM, 2021.

[32] Elette Boyle, Ran Cohen, and Aarushi Goel. Breaking the o($\surd$ n)-bit barrier: Byzantine agreement with polylog bits per party. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 319–330. ACM, 2021.

[33] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[34] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[35] Sophie JC Caron, Vanessa Ruta, Larry F Abbott, and Richard Axel. Random convergence of olfactory inputs in the drosophila mushroom body. *Nature*, 497(7447):113–117, 2013.

[36] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[37] Keren Censor-Hillel and Tariq Toukan. On fast and robust information spreading in the vertex-congest model. *Theoretical Computer Science*, 2017.

[38] Diptarka Chakraborty and Keerti Choudhary. New extremal bounds for reachability and strong-connectivity preservers under failures. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, pages 25:1–25:20, 2020.

[39] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[40] Philippe Chassaing and Lucas Gerin. Efficient estimation of the cardinality of large data sets. *arXiv preprint math/0701347*, 2007.

[41] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[42] Zhiwei Chen and Aoqian Zhang. A survey of approximate quantile computation on large-scale data. *IEEE Access*, 8:34585–34597, 2020.

[43] Bogdan S. Chlebus, Dariusz R. Kowalski, and Jan Olkowski. Fast agreement in networks with byzantine nodes. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, pages 30:1–30:18, 2020.

[44] Chi-Ning Chou, Kai-Min Chung, and Chi-Jen Lu. On the algorithmic power of spiking neural networks. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, pages 26:1–26:20, 2019.

[45] Chi-Ning Chou and Mien Brabeeba Wang. Ode-inspired analysis for the biological version of oja's rule in solving streaming pca. In *Conference on Learning Theory*, pages 1339–1343. PMLR, 2020.

[46] Brian A. Coan and Jennifer L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Inf. Comput.*, 97(1):61–85, 1992.

[47] Ran Cohen, Iftach Haitner, Nikolaos Makriyannis, Matan Orland, and Alex Samorodnitsky. On the round complexity of randomized byzantine agreement. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, pages 12:1–12:17, 2019.

[48] Graham Cormode, Mayur Datar, Piotr Indyk, and Shanmugavelayutham Muthukrishnan. Comparing data streams using hamming norms (how to zero in). *IEEE Transactions on Knowledge and Data Engineering*, 15(3):529–540, 2003.

[49] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[50] Max Dabagia, Santosh S Vempala, and Christos Papadimitriou. Assemblies of neurons learn to classify well-separated distributions. In *Conference on Learning Theory*, pages 3685–3717. PMLR, 2022.

[51] Sanjoy Dasgupta, Timothy C Sheehan, Charles F Stevens, and Saket Navlakha. A neural data structure for novelty detection. *Proceedings of the National Academy of Sciences*, 115(51):13093–13098, 2018.

[52] Sanjoy Dasgupta, Charles F Stevens, and Saket Navlakha. A neural algorithm for a fundamental computing problem. *Science*, 358(6364):793–796, 2017.

[53] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.

[54] Peter Dayan and Laurence F Abbott. Theoretical neuroscience: computational and mathematical modeling of neural systems. *MIT press*, 2005.

[55] RE Lee DeVille and Charles S Peskin. Synchrony and asynchrony in a fully stochastic neural network. *Bulletin of mathematical biology*, 70(6):1608–1633, 2008.

[56] Michael Dinitz and Robert Krauthgamer. Fault-tolerant spanners: better and simpler. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 169–178. ACM, 2011.

[57] Danny Dolev. The byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982.

[58] Danny Dolev, Michael J. Fischer, Robert J. Fowler, Nancy A. Lynch, and H. Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

[59] Danny Dolev and Ezra N. Hoch. Constant-space localized byzantine consensus. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 167–181, 2008.

[60] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 605–617, 2003.

[61] Cynthia Dwork, David Peleg, Nicholas Pippenger, and Eli Upfal. Fault tolerance in networks of bounded degree. *SIAM J. Comput.*, 17(5):975–988, 1988.

[62] Huawei Fan, Yafeng Wang, Hengtong Wang, Ying-Cheng Lai, and Xingang Wang. Autapses promote synchronization in neuronal networks. *Scientific reports*, 8(1):580, 2018.

[63] Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.

[64] Gerald T Finnerty, Michael N Shadlen, Mehrdad Jazayeri, Anna C Nobre, and Dean V Buonomano. Time in cortical circuits. *Journal of Neuroscience*, 35(41):13912–13916, 2015.

[65] Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International Conference on Fundamentals of Computation Theory*, pages 127–140. Springer, 1983.

[66] 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.

[67] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In Michael A. Malcolm and H. Raymond Strong, editors, *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*, pages 59–70. ACM, 1985.

[68] Orr Fischer and Merav Parter. Distributed CONGEST algorithms against mobile adversaries. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 262–273. ACM, 2023.

[69] Mattias Fitzi and Ueli Maurer. From partial consistency to global broadcast. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 494–503, 2000.

[70] Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985.

[71] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *IN AOFA '07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*, 2007.

[72] Steve Furber. Large-scale neuromorphic computing systems. *Journal of neural engineering*, 13(5):051001, 2016.

[73] Chaya Ganesh and Arpita Patra. Broadcast extensions with optimal communication and round complexity. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 371–380, 2016.

[74] Elad Ganmor, Ronen Segev, and Elad Schneidman. Sparse low-order interaction network underlies a highly correlated and learnable neural population code. *Proceedings of the National Academy of sciences*, 108(23):9679–9684, 2011.

[75] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for $n > 3$t processors in $t + 1$ rounds. *SIAM J. Comput.*, 27(1):247–290, 1998.

[76] Timothy J Gawne and Barry J Richmond. How independent are the messages carried by adjacent inferior temporal cortical neurons? *Journal of Neuroscience*, 13(7):2758–2771, 1993.

[77] Wulfram Gerstner, Andreas K Kreiter, Henry Markram, and Andreas VM Herz. Neural codes: firing rates and beyond. *Proceedings of the National Academy of Sciences*, 94(24):12740–12741, 1997.

[78] Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 3–12. ACM, 2015.

[79] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

[80] Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil P. Vadhan. Better pseudorandom generators from milder pseudorandom restrictions. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 120–129, 2012.

[81] Fabrizio Grandoni and Virginia Vassilevska Williams. Faster replacement paths and distance sensitivity oracles. *ACM Trans. Algorithms*, 16(1):15:1–15:25, 2020.

[82] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, 1995.

[83] Simon S Haykin. *Neural networks and learning machines*, volume 3. Pearson, 2009.

[84] Yael Hitron, Nancy A. Lynch, Cameron Musco, and Merav Parter. Random sketching, clustering, and short-term memory in spiking neural networks. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, pages 23:1–23:31, 2020.

[85] Yael Hitron, Cameron Musco, and Merav Parter. Spiking neural networks through the lens of streaming algorithms. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 10:1–10:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[86] Yael Hitron and Merav Parter. Counting to ten with two fingers: Compressed counting with spiking neurons. In *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, pages 57:1–57:17, 2019.

[87] Yael Hitron and Merav Parter. Broadcast CONGEST algorithms against adversarial edges. 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 23:1–23:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[88] Yael Hitron and Merav Parter. General CONGEST compilers against adversarial edges. 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 24:1–24:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[89] Yael Hitron, Merav Parter, and Gur Perri. The computational cost of asynchronous neural communication. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, pages 48:1–48:47, 2020.

[90] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[91] Kaori Ikeda and John M Bekkers. Autapses. *Current Biology*, 16(9):R308, 2006.

[92] Damien Imbs and Michel Raynal. Simple and efficient reliable broadcast in the presence of byzantine processes. *arXiv preprint arXiv:1510.06882*, 2015.

[93] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of the ACM (JACM)*, 53(3):307–323, 2006.

[94] Piotr Indyk and Eric Price. K-median clustering, model-based compressive sensing, and sparse recovery for earth mover distance. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 627–636, 2011.

[95] Piotr Indyk and David P. Woodruff. Tight lower bounds for the distinct elements problem. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 283–288, 2003.

[96] Eugene M Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004.

[97] John Kallaugher and Eric Price. Separations and equivalences between turnstile streaming and linear sketching. In *Symposium on Theory of Computing, STOC 2020*, 2020.

[98] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 41–52, 2010.

[99] Michael Kapralov, Aida Mousavifar, Cameron Musco, Christopher Musco, Navid Nouri, Aaron Sidford, and Jakab Tardos. Fast and space efficient spectral sparsification in dynamic streams. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1833. SIAM, 2020.

[100] David R Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.

[101] Zohar Karnin, Kevin Lang, and Edo Liberty. Optimal quantile approximation in streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 71–78. IEEE, 2016.

[102] Karthik C. S. and Merav Parter. Deterministic replacement path covering. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 704–723, 2021.

[103] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.

[104] Muhammad Mukaram Khan, David R Lester, Luis A Plana, A Rast, Xin Jin, Eustace Painkras, and Stephen B Furber. Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2849–2856. Ieee, 2008.

[105] Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact byzantine consensus on undirected graphs under local broadcast model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 327–336, 2019.

[106] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 87–98, 2006.

[107] Chiu-Yuen Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 275–282. ACM, 2004.

[108] Fabian Kuhn, Joel Spencer, Konstantinos Panagiotou, and Angelika Steger. Synchrony and asynchrony in neural networks. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete algorithms*, pages 949–964. SIAM, 2010.

[109] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[110] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[111] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10:508, 2016.

[112] Robert A. Legenstein, Wolfgang Maass, Christos H. Papadimitriou, and Santosh S. Vempala. Long term memory and the densest k-subgraph problem. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, pages 57:1–57:15, 2018.

[113] Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling ino (congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

[114] Yi Li, Huy L. Nguyen, and David P. Woodruff. Turnstile streaming algorithms might as well be linear sketches. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 174–183, 2014.

[115] Yi Li and David P. Woodruff. A tight lower bound for high frequency moment estimation with small error. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, pages 623–638, 2013.

[116] Andrew C Lin, Alexei M Bygrave, Alix De Calignon, Tzumin Lee, and Gero Miesenböck. Sparse, decorrelated odor coding in the mushroom body enhances learned odor discrimination. *Nature neuroscience*, 17(4):559–568, 2014.

[117] Benjamin Lindner. Some unsolved problems relating to noise in biological systems. *Journal of Statistical Mechanics: Theory and Experiment*, 2009(01):P01008, 2009.

[118] Nancy Lynch, Cameron Musco, and Merav Parter. Computational tradeoffs in biological neural networks: Self-stabilizing winner-take-all networks. *Innovations in Theoretical Computer Science*, 2017.

[119] Nancy Lynch, Cameron Musco, and Merav Parter. Spiking neural networks: An algorithmic perspective. In *5th Workshop on Biological Distributed Algorithms (BDA 2017)*, 2017.

[120] Nancy Lynch and Mien Brabeeba Wang. Brief announcement: Integrating temporal information to spatial information in a neural circuit. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[121] Nancy A. Lynch and Frederik Mallmann-Trenn. Learning hierarchically-structured concepts. *Neural Networks*, 143:798–817, 2021.

[122] Nancy A. Lynch and Frederik Mallmann-Trenn. Learning hierarchically-structured concepts II: overlapping concepts, and networks with feedback. In Sergio Rajsbaum, Alkida Balliu, Joshua J. Daymude, and Dennis Olivetti, editors, *Structural Information and Communication Complexity - 30th International Colloquium, SIROCCO 2023, Alcalá de Henares, Spain, June 6-9, 2023, Proceedings*, volume 13892 of *Lecture Notes in Computer Science*, pages 46–86. Springer, 2023.

[123] Nancy A. Lynch and Cameron Musco. A basic compositional model for spiking neural networks. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, volume 13560 of *Lecture Notes in Computer Science*, pages 403–449. Springer, 2022.

[124] Nancy A. Lynch, Cameron Musco, and Merav Parter. Neuro-ram unit with applications to similarity testing and compression in spiking neural networks. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 33:1–33:16, 2017.

[125] Jun Ma, Xinlin Song, Wuyin Jin, and Chuni Wang. Autapse-induced synchronization in a coupled neuronal network. *Chaos, Solitons & Fractals*, 80:31–38, 2015.

[126] Wolfgang Maass. Lower bounds for the computational power of networks of spiking neurons. *Electronic Colloquium on Computational Complexity (ECCC)*, 1(19), 1994.

[127] Wolfgang Maass. On the computational complexity of networks of spiking neurons. In *Advances in neural information processing systems*, pages 183–190, 1995.

[128] Wolfgang Maass. On the computational power of noisy spiking neurons. In *Advances in neural information processing systems*, pages 211–217, 1996.

[129] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[130] Wolfgang Maass. Neural computation with winner-take-all as the only nonlinear operation. In *Advances in Neural Information Processing Systems 12 (NIPS)*, pages 293–299, 1999.

[131] Wolfgang Maass. On the computational power of winner-take-all. *Neural computation*, 12(11):2519–2535, 2000.

[132] Wolfgang Maass, Christos H. Papadimitriou, Santosh S. Vempala, and Robert Legenstein. Brain computation: A computer science perspective. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State of the Art and*

Perspectives, volume 10000 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2019.

[133] Wolfgang Maass, Georg Schnitger, and Eduardo D Sontag. On the computational power of sigmoid versus boolean threshold circuits. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[134] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD Record*, 27(2):426–435, 1998.

[135] Rajit Manohar and Yoram Moses. The eventual c-element theorem for delay-insensitive asynchronous circuits. In *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 102–109. IEEE, 2017.

[136] Danijela Marković, Alice Mizrahi, Damien Querlioz, and Julie Grollier. Physics for neuromorphic computing. *Nature Reviews Physics*, 2(9):499–510, 2020.

[137] Alexandre Maurer and Sébastien Tixeuil. On byzantine broadcast in loosely connected networks. In *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, pages 253–266, 2012.

[138] Hugo Merchant, Deborah L Harrington, and Warren H Meck. Neural basis of the perception and estimation of time. *Annual review of neuroscience*, 36:313–336, 2013.

[139] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[140] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[141] Michael G Müller, Christos H Papadimitriou, Wolfgang Maass, and Robert Legenstein. A model for structured information representation in neural networks of the brain. *eneuro*, 7(3), 2020.

[142] J Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[143] Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[144] Nandakumar S Narayanan, Eyal Y Kimchi, and Mark Laubach. Redundancy and synergy of neuronal ensembles in motor cortex. *Journal of Neuroscience*, 25(17):4207–4216, 2005.

[145] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, pages 28:1–28:17, 2020.

[146] Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, 1994.

[147] Christos H. Papadimitriou and Angela D. Friederici. Bridging the gap between neurons and cognition through assemblies of neurons. *Neural Comput.*, 34(2):291–306, 2022.

[148] Christos H. Papadimitriou and Santosh S. Vempala. Random projection in the brain and computation with assemblies of neurons. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, pages 57:1–57:19, 2019.

[149] Christos H Papadimitriou, Santosh S Vempala, Daniel Mitropolsky, Michael Collins, and Wolfgang Maass. Brain computation by assemblies of neurons. *Proceedings of the National Academy of Sciences*, 117(25):14464–14472, 2020.

[150] Michel Paquette and Andrzej Pelc. Fast broadcasting with byzantine faults. *Int. J. Found. Comput. Sci.*, 17(6):1423–1440, 2006.

[151] Ojas Parekh, Yipu Wang, Yang Ho, Cynthia Phillips, Ali Pinar, James Aimone, and William Severa. Neuromorphic graph algorithms. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States);, 2021.

[152] Merav Parter. Small cuts and connectivity certificates: A fault tolerant approach. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[153] Merav Parter and Eylon Yogev. Congested clique algorithms for graph spanners. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[154] Merav Parter and Eylon Yogev. Distributed algorithms made secure: A graph theoretic approach. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1693–1710. SIAM, 2019.

[155] Merav Parter and Eylon Yogev. Low congestion cycle covers and their applications. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1673–1692, 2019.

[156] Merav Parter and Eylon Yogev. Optimal short cycle decomposition in almost linear time. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 89:1–89:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[157] Merav Parter and Eylon Yogev. Optimal short cycle decomposition in almost linear time. `https://www.weizmann.ac.il/math/parter/sites/math.parter/files/uploads/main-icalp-cycles-full.pdf`, 2019.

[158] Merav Parter and Eylon Yogev. Secure distributed computing made (nearly) optimal. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 107–116, 2019.

[159] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[160] Andrzej Pelc. Reliable communication in networks with byzantine link failures. *Networks*, 22(5):441–459, 1992.

[161] Andrzej Pelc. Fault-tolerant broadcasting and gossiping in communication networks. *Networks: An International Journal*, 28(3):143–156, 1996.

[162] Andrzej Pelc and David Peleg. Broadcasting with locally bounded byzantine faults. *Inf. Process. Lett.*, 93(3):109–115, 2005.

[163] Andrzej Pelc and David Peleg. Feasibility and complexity of broadcasting with random transmission failures. *Theoretical Computer Science*, 370(1-3):279–292, 2007.

[164] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. SIAM, 2000.

[165] Christoph Pokorny, Matias J Ison, Arjun Rao, Robert Legenstein, Christos Papadimitriou, and Wolfgang Maass. Stdp forms associations between memory traces in networks of spiking neurons. *Cerebral Cortex*, 30(3):952–968, 2020.

[166] Jason L Puchalla, Elad Schneidman, Robert A Harris, and Michael J Berry. Redundancy in the population code of the retina. *Neuron*, 46(3):493–504, 2005.

[167] Michel Raynal. Consensus in synchronous systems: A concise guided tour. In *2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings.*, pages 221–228. IEEE, 2002.

[168] Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Procceedings 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.

[169] Nicola Santoro and Peter Widmayer. Time is not a healer. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 304–313. Springer, 1989.

[170] Nicola Santoro and Peter Widmayer. Distributed function evaluation in the presence of transmission faults. In *Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 16-18, 1990, Proceedings*, pages 358–367, 1990.

[171] Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2-3):232–249, 2007.

[172] Johannes Schemmel, Daniel Brüderle, Andreas Grübl, Matthias Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1947–1950. IEEE, 2010.

[173] Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

[174] Ulrich Schmid, Bettina Weiss, and John Rushby. Formally verified byzantine agreement in presence of link faults. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 608–616. IEEE, 2002.

[175] Elad Schneidman, Michael J Berry, Ronen Segev, and William Bialek. Weak pairwise correlations imply strongly correlated network states in a neural population. *Nature*, 440(7087):1007–1012, 2006.

[176] Elad Schneidman, William Bialek, and Michael Ii. An information theoretic approach to the functional classification of neurons. *Advances in neural information processing systems*, 15, 2002.

[177] William Severa, Rich Lehoucq, Ojas Parekh, and James B. Aimone. Spiking neural algorithms for markov process random walk. In *2018 International Joint Conference on Neural Networks, IJCNN 2018, Rio de Janeiro, Brazil, July 8-13, 2018*, pages 1–8. IEEE, 2018.

[178] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449, 1992.

[179] Hin-Sing Siu, Yeh-Hao Chin, and Wei-Pang Yang. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):335–345, 1998.

[180] Jens Sparsø. Asynchronous circuit design-a tutorial. In *Chapters 1-8 in" Principles of asynchronous circuit design-A systems Perspective"*. Kluwer Academic Publishers, 2001.

[181] Daniel A. Spielman. Lecture notes in spectral graph theory, November 2009.

[182] Lili Su, Chia-Jung Chang, and Nancy Lynch. Spike-based winner-take-all computation: Fundamental limits and order-optimal circuits. *Neural Computation*, 31(12):2523–2561, 2019.

[183] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, 2019.

[184] Sam Toueg, Kenneth J Perry, and TK Srikanth. Fast distributed agreement. *SIAM Journal on Computing*, 16(3):445–457, 1987.

[185] Thomas P. Trappenberg. Fundamentals of computational neuroscience (2. ed.). *Oxford University Press*, 2009.

[186] Misha V Tsodyks and Henry Markram. The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. *Proceedings of the national academy of sciences*, 94(2):719–723, 1997.

[187] Eli Upfal. Tolerating a linear number of faults in networks of bounded degree. *Inf. Comput.*, 115(2):312–320, 1994.

[188] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

[189] Leslie G Valiant. Circuits of the mind. *Oxford University Press on Demand*, 2000.

[190] Leslie G Valiant. A neuroidal architecture for cognitive computation. *Journal of the ACM (JACM)*, 47(5):854–882, 2000.

[191] Leslie G Valiant. Memorization and association on a realistic neural model. *Neural Computation*, 17(3):527–555, 2005.

[192] Leslie G. Valiant. Capacity of neural networks for lifelong learning of composable tasks. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 367–378, 2017.

[193] Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms (TALG)*, 9(2):1–13, 2013.

[194] David P. Woodruff. Optimal space lower bounds for all frequency moments. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 167–175, 2004.

[195] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1130–1143, 2017.