FAIRNESS FOR DISTRIBUTED ALGORITHMS

A Dissertation
submitted to the Faculty of the
Graduate School of Arts and Sciences
of Georgetown University
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in Computer Science

By

Tonghe Wang, M.S.

Washington, DC
July 25, 2017

FAIRNESS FOR DISTRIBUTED ALGORITHMS

Tonghe Wang, M.S.

Dissertation Advisor: Calvin Newport, Ph.D.

ABSTRACT

In this thesis, we explore a new but intuitive notion of "fairness" that focuses on the equality of outcomes of distributed algorithms. In particular, this thesis studies new definitions of fairness with respect to four general topics: graph algorithms, communication algorithms, contention resolution algorithms, and consensus algorithms.

First, we explore fair graph algorithms by tackling with two different problems: maximal independent set and vertex coloring. We propose a novel definition of fairness that applies to each problem and study new upper and lower bounds with respect to this metric. This new definition of fairness requires, roughly speaking, that each node has a similar probability of ending up with each possible outcome.

We then turn our attention to communication algorithms that consider the quality of the communication links. We explore new upper and lower bounds for fair rate selection algorithms that guarantee throughput within reasonable factors of the optimal achievable rate at each receiver. We study both single hop and multihop networks, as well as links with static quality and those with qualities that can change over time.

We then consider new fair solutions to the classical contention resolution problem, where we define "fairness" in this context to mean each process has a similar probability of being the process that first breaks symmetry. We focus in particular on the setting with multiple channels and collision detection (for which little results were known until recently). We describe and analyze a new fair contention resolution algorithm that comes within small factors of matching a recently proved lower bound of this setting [83]. Of equal importance,

our solutions introduce a novel new technique in which we leverage a distributed structure we call *coalescing cohorts* to simulate a well-known parallel search strategy from the structured PRAM CREW model [98] in our unstructured distributed model.

Finally, we turn our attention to the asynchronous shared memory model and study the new concept of fair consensus. A fair consensus algorithm, intuitively, guarantees every proposed value is decided with similar probability. Defining and achieving this property is more complex. We describe and analyze a new consensus algorithm that satisfies our new definition of fairness. We also show how to implement a replicated state machine that applies a fair consensus algorithm as subroutine.

INDEX WORDS: Fairness, Lower bound, Maximal independent set, Inequality factor, Symmetry breaking, Coloring, Bitrate, Multicast, Packet transmission, Latency, Competitive ratio, Static, Fading, Order preserving, Contention resolution, Collision detection, Asynchronous shared memory, Consensus, Stochastic scheduler, Fault-tolerant, Replicated state machine

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

CHAPTER 1

INTRODUCTION

The term "fairness" in the study of distributed algorithms was defined by Lynch *et al.* [80, 81] to constrain the scheduling of computational processes in an asynchronous system. This definition of fairness is characterized as "eventuality" [31]: a fair execution must continuously provide every process an opportunity to take a step, preventing any process from being starved forever. Fairness of this type has also been defined for specific problems in the study of distributed algorithms. The most well-known example is the classical mutual exclusion problem, in which multiple processes compete for exclusive access to a shared resource. Strong solutions to this problem [22, 23, 25] guarantee that every process attempting to get access to the resource will eventually succeed—preventing unfair starvation by other processes. An even stronger constraint satisfied by some mutual exclusion algorithms is to bound the time a process might have to wait before succeeding.

In many contexts, however, "fairness" primarily means "equality", a goal that is more difficult to achieve than "eventuality". This thesis explores new notions of fairness in the study of distributed algorithms that focus on the equality of outcomes. Generally speaking, a fair algorithm, in this context, gives each object about the same chance to win the relevant competition specified by the problem definition. For example, in studying mutual exclusion, existing definitions of fairness often require that every competing process eventually gets access to the resource, whereas the types of definitions we study might require that every competing process has a similar probability of gaining access to the resource. In particular, this thesis studies this new form of fairness with respect to four general topics—graph

algorithms, communication algorithms, contention resolution algorithms, and consensus algorithms—which we summarize below.

1.1 Graph Algorithms

Generally speaking, a distributed algorithm is called a graph algorithm if one process is assigned to each node in a graph and these processes work together to solve a problem from graph theory with respect to this graph. Such algorithms are well-studied in the distributed algorithms literature and have many applications [9, 14, 16, 17, 27, 38, 43, 48, 50, 54, 58, 59, 73, 76, 77, 78, 82, 85, 86, 97, 105]. In this thesis, we will study the problem of achieving fairness with distributed graph algorithms.

We begin by studying the maximal independent set (MIS) problem. The goal of an MIS algorithm is to select a subset of the graph node set such that every node is covered by this set but no two nodes in this set are neighbors. We propose the concept of "inequality factor" as a fairness metric for randomized MIS algorithms. The inequality factor measures the greatest gap of the probabilities to be selected as MIS nodes between different nodes. We call a randomized MIS algorithm *fair* if its inequality factor is no greater than a constant. This metric is non-trivial given that existing MIS algorithms might not be fair. Luby's MIS algorithm [78], for example, has large inequality factors in certain graphs. On the other hand, this thesis describes and analyzes new MIS algorithms that are fair in trees (both rooted and unrooted), bipartite graphs, planar graphs, and other common graph classes. We list the inequality factors and the time bounds (that hold with high probability) for different MIS algorithms in Table 1.1, where n is the number of nodes in the network, ϵ is the probability that some process falls into the fourth stage of our algorithm, and $f(n, k)$ is the time complexity of some k-coloring algorithm we use as a subroutine of our algorithm.

Table 1.1: Inequality factor and time complexity guarantees for the fair MIS algorithms studied in this thesis. Each algorithm works only for the specified graph family (see Chapter 3 for details).

Graph family	**Algorithm**	**Inequality factor**	**Time complexity**
Rooted trees	FAIRROOTED	≤ 4	$O(\log^* n)$
Unrooted trees	FAIRTREE	$\leq 4/(1-\epsilon)$	$O(\log n)$
Bipartite graphs	FAIRBIPART	≤ 8	$O(\log^2 n)$
k-colorable graphs	COLORMIS	$O(k)$	$O(f(n,k)+\log^2 n)$

Note that these time bounds are within logarithmic factors of the best known non-fair MIS results for these graph classes (see Chapter 2).

Distributed vertex coloring is another classical graph problem. To conclude our study on fair graph algorithms, we will study fair distributed vertex coloring algorithms. Given a set of k colors, the goal of a vertex k-coloring algorithm, or k-coloring algorithm for short, is to have each node output one color from this set in a way that no neighbors output the same color. We introduce a natural new definition of fairness for coloring algorithms. Similar to MIS algorithms, we define the inequality factor of a k-coloring algorithm with respect to a fixed color from the color set as the greatest difference between the probabilities of nodes outputting this color. We say that a k-coloring algorithm is fair if the inequality factors with respect to all k colors are bounded by a constant. We first provide a fast but unfair $(\Delta+1)$-coloring algorithm that completes in $O(\log n)$ rounds with high probability, with n being the number of nodes in the network and Δ being the maximum node degree. Then we give a fair $(\Delta+1)$-coloring algorithm that terminates in $O(\log^2 n)$ rounds with high probability using the unfair algorithm as subroutine.

There are multiple scenarios in which fair graph algorithms would prove useful. In a network monitoring application that has MIS nodes log the behavior of their neighbors, for example, being in the MIS consigns a node to filling up its storage at a higher rate than its non-MIS neighbors. Similarly, in constructing a network backbone, outputting some color consigns a node to processing much more traffic than nodes outputting other colors in the same network. Using a fair MIS or coloring algorithm as a subroutine can ensure that the corresponding workload is more evenly distributed.

1.2 Communication Algorithms

The message passing model used in the study of graph algorithms fails to capture the potential message loss due to poor link quality. This kind of behavior is particularly common in wireless networks, e.g., over the links that are established according to the IEEE 802.11 standard [19]. In this thesis, we study a wireless communication network model that captures the impact of link quality on message transmission. In this model, messages are divided into packets, and at most one packet can be delivered during one single transmission. For each packet, the sender must specify a transmission speed, often called *bitrate*, from multiple available speeds. The quality of a communication link is measured by the speed limit of the link, i.e., the fastest transmission speed supported by this link. Some receivers, for example, might have high quality links that are able to support fast transmission rates with the sender, while others might have low quality links that can only support slow rates. A transmission succeeds (and the receiver receives the packet) if and only if the message is sent at a speed within the speed limit. In other word, if a sender broadcasts a message in its neighborhood at a fast speed, receivers with poor link quality might not be able to receive it. We assume that a sender does not know the quality of the links to nearby

receivers. Algorithms that assist in selecting bitrates for transmissions in this setting are called rate selection (or adaptation) algorithms.

In unicast communication with a single message destination in the network, feedback information on packet transmissions, for example packet loss information, is often used to test different rates and then find the fastest successful rate [49, 52, 60, 74]. This thesis, however, focuses on the multicast problem that requires a sender to deliver information to multiple nearby receivers. We do not assume the availability of acknowledgement strategies because they can result in traffic congestion if multiple receivers attempt to acknowledge a packet at the same time. That is to say, the sender of a message is not aware of the fate of packet transmission, and corresponding rate selection algorithms are said to be *blind*.

Existing blind rate selection algorithms usually attempt to limit frame loss by adopting more reliable, and therefore slower bitrates (for instance Holland *et al.* [49]). However, these strategies are *unfair* for a receiver with a higher quality link, for packets could have arrived at a much faster speed. In this thesis, we seek fair multicast protocols that can ensure each receiver gets packets delivered with throughput comparable to the best throughput possible given the quality of its individual link. The fairness of a rate selection algorithm is evaluated by its *competitive ratio* of its *(expected) average latency*, i.e., the (expected) average arriving time of packets at each receiver, with respect to that of the optimal solution in the same setting.

This thesis analyzes a series of new rate selection algorithms for blind multicast. The first setting in which we study these algorithms is the single hop network with static links. In this setting, the sender has a directed link to each receiver in the network, and the quality of each link does not change over time. We describe and analyze new randomized and deterministic rate selection algorithms that achieve small competitive ratios compared to optimal. We prove these results (essentially) tight with a lower bound on achievable com-

Table 1.2: Upper and lower bounds for the competitive ratio of blind rate selection algorithms (see Chapter 4 for details)

Network Type	**Link Type**	**Algorithm Type**	**Upper Bound**	**Lower Bound**
Single hop	Static	Randomized	$O(\log L)$	$\Omega\left(\frac{\log L}{\log\log L}\right)$
		Deterministic		
Multihop	Static	Deterministic	$O(\log L)$	$\Omega\left(\frac{\log L}{\log\log L}\right)$
Single hop	Fading	Deterministic	$O(L)$	$\Omega(L)$
		Randomized		$\Omega(\sqrt{L})$

petitive ratios in this setting. We conclude, however, by proving that no rate selection can be competitive if link quality can vary from round to round in the fading model.

We then turn our attention to studying multihop networks, where the network topology is no longer necessarily a clique. We describe and analyze a new algorithm that is fair with respect to the best single path for each receiver, but we then prove that practical algorithms cannot be fair with respect to the optimal solution achieved by multiplexing traffic over multiple paths.

Table 1.2 summaries our upper and lower bound results for blind rate selection algorithms, where L is the maximum latency, i.e., the number of rounds required to send a packet with the slowest bitrate.

Fair rate selection algorithms are important in practice, as fair algorithms prevent the senders from having to default to a slow rate to be safe: receivers with high quality links can enjoy high throughput.

1.3 CONTENTION RESOLUTION ALGORITHMS

The contention resolution problem assumes an unknown subset of n possible processes that are *activated* and connected to a shared multiple-access channel (MAC). The problem is solved in the first round that one of these *active* processes transmits alone on the shared channel. Contention resolution (which is also sometimes called *leader election* or *wake-up*, depending on the model assumptions and context) is one of the longest studied problems in the theory of distributed algorithms. It was first identified in the 1970 paper on the ALOHA radio network and variations have been extensively explored by theoretical computer scientists and mathematicians ever since; e.g., [7, 20, 36, 39, 42, 51, 53, 56, 83, 92, 102].

Contention resolution has been studied under many different model variations. Two of the most important assumptions concern collision detection and number of available channels. Tight bounds are known for contention resolution on a single channel with and without collision detection [26, 51, 83, 102], and for contention resolution on multiple channels without collision detection [21, 83]. Until recently, however, little was known about the remaining case of multiple channels with collision detection. In 2014 [83], progress was made with a new lower bound that established $\Omega\left(\frac{\log n}{\log \mathcal{C}} + \log\log n\right)$ rounds are needed to solve contention resolution with collision detection and $\mathcal{C} \geq 1$ channels (with high probability in n). This bound holds *even* if we consider the easier restricted case where only two processes out of the n possible ones will be activated. In [83], the question of this bound's optimality was left as an open problem—though it was suspected to be *not tight* because it followed from the limits of a specific proof technique and had no connection to a particular upper bound intuition.

In this thesis, we resolve this open problem with a solution that we also prove satisfies a natural definition of fairness. In more detail, we show (perhaps surprisingly) that the lower bound from [83] *is tight* or within a $\log\log\log n$ factor of tight for all relevant values

of n and $\mathcal{C}$. To do so, we describe and analyze two new contention resolution algorithms for MACs with collision detection and $\mathcal{C} > 1$ channels. The first algorithm (presented in Section 5.2) solves the problem in an optimal $O\left(\frac{\log n}{\log \mathcal{C}} + \log\log n\right)$ rounds in the restricted case where only two processes are activated—exactly matching the lower bound from [83], which applies to this restricted case. The second algorithm solves the problem in a near optimal $O\left(\frac{\log n}{\log \mathcal{C}} + (\log\log n)(\log\log\log n)\right)$ rounds for the general case where any number of processes are activated—falling just shy of the lower bound by a $\log\log\log n$ factor when $\mathcal{C}$ is large (i.e., the $\log\log n$ term dominates) and matching the lower bound otherwise. We prove that both algorithms are fair in the sense that each active process has the same probability to be selected as the leader (i.e., the process that first transmits alone). This notion of fairness is useful in practice as the "leader" that breaks contention in these low-level algorithms might then be required to perform work on behalf of the network. Fair contention resolution guarantees over time (and therefore over multiple elections) that this work is well spread out.

1.4 Consensus Algorithms

For the final topic explored in this thesis, we turn our attention to the asynchronous shared memory model. One of the fundamental topics in distributed computation in this model is the consensus problem [69]. In this problem, each process receives as input an initial value and they attempt to agree on a common value from among these inputs. Consensus algorithms have a variety of practical applications, such as providing synchronization for an asynchronous system [62, 63, 96], implementing a replicated state machine in a client-server system [57, 62, 64], and crash detection and recovery from failure in a distributed system [13, 41, 68].

Standard consensus algorithms guarantee that some initial value is eventually decided, but make no guarantees about which value. In this thesis, we study fair consensus algorithms that guarantee that processes have roughly equal chance to have their initial value decided. We also explore how consensus algorithms with this property can then be applied as subroutines to add fairness to higher level applications, by studying new fair replicated state machine implementations [6, 33, 57, 62, 64, 88].

In more detail, we start by identifying definitions of fairness that can accommodate the reality that processes can crash, i.e., stop taking steps in the future. If a process crashes early before it can propose its initial value, its value likely is not in contention to be decided. Our definitions of fairness accommodate this reality. Then we describe and analyze new fair consensus algorithms. To accomplish this, we take a black box approach: we demonstrate how to transform a generic fault-tolerant consensus algorithm (treated as a black box) into a fair consensus algorithm. Our black box transformation will depend on extra assumptions made of the scheduler that prevent arbitrarily long starvation of process steps. Note that for completeness, we also show how to implement a consensus algorithm using these same assumptions used by our transformation, which is potentially of standalone interest.

As mentioned, a main application of consensus algorithms is implementing replicated state machines, largely applied in distributed database design [6, 33, 88]. These implementations often rely on consensus algorithms to synchronize the order in which operations are applied to the state machine state replicated over multiple processes. Existing replicated state machine algorithms use complicated logic to ensure all operations get a chance to be applied [46, 62]. We explore how to use fair consensus algorithms as a subroutine to achieve these properties with simpler replicated sate machine algorithms.

1.5 IMPACT

This thesis has two main types of contributions. The first is new concrete bounds that achieve our new notion of fairness in different contexts. The second is the general idea that equality of outcomes is something that is useful and worth studying in the field of distributed algorithms.

CHAPTER 2

RELATED WORK

In this chapter, we will survey relevant related work for each of the problems we study. Note that this related work survey will include our own existing published work on the topics tacked in the thesis [27, 28, 29, 37].

2.1 RELATED WORK ON GRAPH ALGORITHMS

This thesis describes and analyzes new fair graph algorithms for the maximal independent set (MIS) and vertex coloring problems. We previously published our results for the MIS problem in [27, 28]. The vertex coloring results are new to this thesis.

We begin by surveying the relevant related work on the MIS problem. Luby's distributed MIS algorithm [78] is the most cited work in the study of this problem. It runs in $O(\log n)$ rounds (with high probability in the network size n) on a general graph. MIS algorithms can run faster if there are restrictions on the class of graphs considered. For example, an $O(\sqrt{\log n \log\log n})$ algorithm is proposed for trees [73], an $O(\log^* n)$ algorithm for growth-bounded graphs [97], and an $o(\log n)$ algorithm for low-degree graphs [9]. Harris *et al.* [44] study average degrees of graphs, and they prove that a graph of a high average degree requires significant time to calculate an MIS. The best known lower bound for general graphs is $\Omega(\sqrt{\log n})$ [59]. These existing results offer no guarantees with respect to fairness, and in some cases it is straightforward to derive example graphs where the algorithm is clearly not fair.

We now survey the relevant related work on the vertex coloring problem. It was proved in the 1970s that deciding the chromatic number of a graph is NP-complete [34, 35]. Famously, however, Appel *et al.* [3] prove that every planar graph is 4-colorable. In fact, planar graphs are an example of a "low-arboricity" graph. More generally, a graph with arboricity of α can be $(\lfloor \alpha(2+\epsilon) \rfloor + 1)$-colored in $O(\alpha \log n)$ time with a *distributed* algorithm [8]. Leveraging the fact that a graph is always $(\Delta + 1)$-colorable, most of the work on distributed vertex coloring searches for efficient $(\Delta + 1)$-coloring algorithms (as oppose to achieving the minimum possible number of colors for the graph). In general graphs, a distributed $(\Delta + 1)$-coloring can be obtained in $O(\log n)$ rounds with high probability by a randomized strategy [76]. Deterministically, this can be completed within $2^{O(\log n)}$ rounds [86]. As for a constant colorable graph, $(\Delta + 1)$-coloring can be completed in $O(\log^* n)$ rounds [17, 38]. In [38], it also shows how to color graphs with smaller Δ in $O(\Delta^2 + \log^* n)$ rounds.

2.2 Related Work on Communication Algorithms

This thesis describes and analyzes new wireless multicast algorithms that attempt to select transmission rates in a manner that fairly makes use of the bandwidth available to each receiver. We published preliminary versions of these results in [37]. Here we survey relevant related work on wireless communication.

The multicast problem has been well studied in wired network settings; see, for example, Harte's book "Introduction to Data Multicasting" [45]. In the wireless setting, multicast techniques are more complicated due to many factors. Chandra *et al.* [15] implement wireless multicast by first sending the multicast data using unicast packets to a single member of the multicast group, and then having other members listening for these packets in promiscuous mode. Sun *et al.* [100] cluster nodes, and the leader of each cluster is

elected by the sender to determine the fate of packets (see [12] for a more detailed survey on wireless multicast). These strategies, however, focus mainly on packet loss detection in order to schedule retransmission, and they do not offer any guarantee about the fairness of the rate at which different receivers receive packets. For example, in these existing solutions, if some receivers have slow connections and some have fast, it is likely that the sender will transmit primarily at the slow speed.

In the study of unicast communication (a single receiver) most rate adaption protocols rely on transmission acknowledgement. Many rate selection algorithms are performed with the help of packet loss information [49, 52, 60, 74]. There are also strategies attempting to detect channel quality [11, 55, 87, 103]. Another approach depends on encoding data in such a way that it can be decoded at the receiver close to the optimal rate for its channel, for example Spinal codes [89] or Strider codes [40].

2.3 Related Work on Contention Resolution

This thesis describes and analyzes new fair contention resolution algorithms. We published these results in [29]. Here we survey relevant related work on contention resolution.

The study of contention resolution developed out of the network random access method introduced in the ALOHA paper in 1970 [1]. Early work on this problem focused on the *stability* of contention resolution algorithms for different packet arrival rates; e.g., [42, 53, 92] (see [32] for a good survey). By the mid-1980's, however, the literature split, with an increasingly number of mathematicians and computer scientists studying a one-shot version of the problem where a single set of packets arrives at the beginning of the execution, and the problem is solved once the first packet is successfully delivered e.g., [39, 56, 102]. This is the version of the problem we study in this thesis.

Contention resolution has been considered under many model assumptions. The two assumptions relevant to this thesis are collision detection capability and the number of available channels. Many of the original papers on contention resolution (e.g., [39, 56, 102]) assume *collision detection*, which they define (as do we) such that *all* nodes learn of a collision during any round with two or more transmitters. A straightforward algorithm solves contention resolution in $O(\log n)$ rounds in this setting with probability 1: active processes use collisions to guide a descent through a binary search tree over the n possible IDs to identify the smallest ID of an active process. This solution is optimal for solving this problem with high probability in n [83]. Without collision detection, by contrast, contention resolution can be solved with high probability in $O(\log^2 n)$ rounds. Jurdzinkski et al. [51] proved this near optimal for uniform[1] algorithms and Colton et al. [26] eliminated the remaining gap. Recently, Newport [83] proved this bound optimal for all algorithms.

Providing processes additional channels also speeds up contention resolution. Daum et al. [21] show how to solve contention resolution with high probability in $O\Big(\log^2 n/\mathcal{C} + \log n\Big)$ rounds with $\mathcal{C} \geq 1$ channels—achieving a linear speed up in $\mathcal{C}$ until reaching the lower limit of $\Omega(\log n)$. In [21], this bound was proved near tight for uniform algorithms. Newport [83] subsequently proved it tight for all algorithms. This same paper proved that solving contention resolution with high probability with collision detection and multiple channels requires $\Omega\Big(\frac{\log n}{\log \mathcal{C}} + \log\log n\Big)$ rounds for all algorithms, even if we consider the restricted case where only two processes are activated. At the time, the best known upper bound for these assumptions required $O(\log n)$ rounds (i.e., the optimal algorithm for collision detection on a single channel). Finally, we note the parallel binary search strategy we

[1] This is one of several terms used to describe a restriction common in many contention resolution lower bounds: the transmission probabilities used by the algorithm are fixed at each process in advance. This restriction forbids, for example, nodes to base their transmission probability in one round on coin flips from an earlier round.

leverage in our new upper bound for general network size came from Snir's classic 1985 paper on parallel searching [98].

2.4 Related Work on Consensus

This thesis describes and analyzes new techniques for deriving fair consensus algorithms. These results are new and have not been previously published. Here we survey the relevant related work.

The consensus problem has been one of the most popular topics in distributed computing. It has been applied to may problems in this setting, including consistency maintenance and crash recovery of distributed database systems [24, 71, 75, 93]. The study of consensus usually assume two types of failures that can affect the processes: the stopping failures (sometimes called crash failures) that prevent a process from taking any further step [94], and Byzantine failures that make a process behave arbitrarily [70]. Lynch *et al.* prove in [30] that deterministic consensus is impossible to solve in the asynchronous shared memory model with read/write registers and a single stopping failure. Later research therefore aims to solve consensus with some extra assumptions on the underlying models. For example, the Paxos algorithm solves consensus and tolerates non-Byzantine failures with the assumption of a leader election strategy to elect a distinguished process that can coordinate the decision [66]. Authorizing the elected leader the access to log entries exclusively, Raft [84] consensus algorithm is much easier to understand and implement in practice compared to Paxos, but it too relies on the ability to elect a leader. Note that another additional assumption that makes consensus possible with stopping failures is assuming randomized algorithms [4, 10, 15, 91]. Some of our results build on similar strategies.

Lamport first describes the implementation of state machines using consensus in [61] to build a fault tolerant system. Much of the recent work on implementing state machine

replication applies Paxos as a subroutine [57, 64, 66, 67, 72]. Lamport also proves that if there are at most f failures in the system, then only $f + 1$ replicas are needed to tolerate stopping failure, and $2f + 1$ replicas are required for Byzantine failure [65]. With the help of state machine replication, database replication is shown to have high-performance and fault-tolerance [6, 33, 88].

Note that some of the solutions we study for fair consensus will leverage constrained schedulers. Alistarh et al. [2] explain the wait-free behavior, observed in practice, of algorithms proved in theory to be only lock-free, by introducing the stochastic scheduler formalism, which provides some minimum probability for each correct process to take the next step. This constraint is minor and matches properties of real systems [2]. Nevertheless, it is useful for avoiding strong impossibility results. We will use a similar constrained scheduler in our study of fair consensus.

CHAPTER 3

FAIR GRAPH ALGORITHMS

In this chapter we study fair versions of distributed graph algorithms. We will focus in particular on graph algorithms in the classical $\mathcal{CONGEST}$ model [90], which divides time into synchronous communication rounds, and allows processes to send one bounded size message on each adjacent link in each round. Performance of a graph algorithm is often measured by its time complexity, i.e. the number of communication rounds required to complete the computation. Graph algorithms are often used as subroutines in higher-level applications, such as network backbone construction [50, 54, 58, 105], where "winning the competition" to be part of the relevant graph structure constructed by the algorithm might consign the process to more work, such as monitoring links or routing network traffic. This thesis studies fairness as a new property of graph algorithms that compares the probabilities for different processes to win the competition. A fair graph algorithm as subroutine might ensure that the corresponding workload is more evenly distributed.

We will describe and analyze new fair distributed graph algorithms for the following two problems: maximal independent set and vertex coloring. In order to measure fairness, we introduce the metric called *inequality factor* that describes the maximum ratio of probabilities for processes to win a competition specified by different problems. A smaller inequality factor indicates more fairness for the algorithm, as processes have similar probabilities of winning. We will explore the connection between fairness and time complexity, as well as the connection between fairness and the underlying properties of the graph.

3.1 MODEL

Our study of fair graph algorithms assumes the classical $\mathcal{CONGEST}$ model [90]. Specifically, the network is described as an undirected graph $G = (V, E)$, with $n = |V|$ being the vertex number. A computation process (also called a "node" in this chapter) is assigned to each vertex in the graph. The edges correspond to communication links. Execution proceeds in synchronous rounds. In each round, each node can reliably send a message to each of its neighbors in the network topology graph. Message size is bounded to $O(\log n)$ bits. Without the loss of generality, each node will have a unique identifier with a bounded size of $O(\log n)$ so that it fits into messages. Each node knows its own ID, its immediate neighbors' IDs, and n, but no other information about the graph or participating processes is provided in advance.

In this chapter, we define $G(U)$ for $U \subseteq V$ to describe the subgraph of G induced by set U, including vertices in U and all edges of G whose endpoints are both in U. We use $N(u)$ for vertex $u \in V$ to denote the neighbor set of u in G. We define $d_G(u, v)$ (or $d(u, v)$ when G is clear from context) to describe the length of the shortest path from u to v in G, and use $D(G) = \max_{u,v \in V} d_G(u, v)$ to denote the diameter of G. A property is said to hold with high probability (*w.h.p.*) when the property holds with a probability at least $1 - 1/n^c$ for a constant $c \geq 1$.

3.2 Fair Maximal Independent Set (MIS) Algorithms

The maximal independent set problem is a classical problem in graph theory. Formally, a maximal independent set of a graph is defined as follows:

Definition 3.2.1 (Maximal independent set (MIS))**.** *Fix some undirected graph $G = (V, E)$. Suppose $I \subseteq V$ and $I \neq \emptyset$. Then I is an independent set of G if for all $x, y \in I$ and $x \neq y$, x and y are not adjacent. An independent set I is maximal if for any $z \in V \setminus I$, $I \cup \{z\}$ is not an independent set.*

We say a distributed algorithm $\mathcal{A}$ computes an MIS in network graph $G = (V, E)$, if the algorithm has every node eventually output a 1 or 0, and the set $I \subseteq V$ of nodes that output 1 define an MIS over G.

Luby's distributed MIS algorithm computes an MIS in $O(\log n)$ rounds (*w.h.p.*) for general graphs [78] (in this section, we call this algorithm Luby). The deterministic algorithm described in [86] solves the MIS problem in $2^{O(\sqrt{\log n})}$ rounds. As for restricted graph families, a faster algorithm with time complexity of $O(\sqrt{\log n \log\log n})$ is designed specially for trees [73]. The best know lower bound for MIS algorithms is $\Omega(\sqrt{\log n})$ [59].

3.2.1 Definition of Fairness for the MIS Problem

Our new definition of fairness for MIS algorithms leverages the observation that a randomized MIS algorithm might treat nodes differently depending on their location in the graph. Luby, for example, offers more opportunities to high-degree nodes to be selected into the final MIS. Intuitively, this difference in probability of joining the MIS is unfair.

We formally define the term in *inequality factor* of a fixed (randomized) MIS algorithm as the maximum ratio of the probabilities of every pair of nodes to be selected in an MIS. In more detail, let $P_{\mathcal{A},G} : V \to [0, 1]$ be the probability that $v \in V$ is selected as an MIS node

of graph G by a fixed MIS algorithm $\mathcal{A}$. The inequality factor of $\mathcal{A}$ in G can be defined as follows [27, 28]:

Definition 3.2.2. *The inequality factor of an MIS algorithm $\mathcal{A}$ w.r.t. a graph G is defined as*

$$F_{\mathcal{A}}(G) = \max_{u,v \in V} \left\{ \frac{P_{\mathcal{A},G}(u)}{P_{\mathcal{A},G}(v)} \right\}.$$

If there exists $v \in V$ such that $P_{\mathcal{A},G}(v) = 0$, then we fix $F_{\mathcal{A}}(G) = \infty$. We also use $F_{\mathcal{A}}(\mathcal{G}) = \max_{G \in \mathcal{G}} F_{\mathcal{A}}(G)$, to denote the worst-case inequality factor of $\mathcal{A}$ w.r.t. a graph family $\mathcal{G}$.

We now use the definition of inequality factor as part of our definition of fairness.

Definition 3.2.3. *For a graph family $\mathcal{G}$, we say an MIS algorithm $\mathcal{A}$ is fair w.r.t. $\mathcal{G}$ if there exists a constant c s.t. $F_{\mathcal{A}}(\mathcal{G}) \leq c$.*

The smallest possible value of $F_{\mathcal{A}}$ is 1, indicating that algorithm $\mathcal{A}$ is *perfectly fair*, i.e., all nodes have exactly the same chance of joining MIS.

Consider the case that G is a star centered on $u_1 \in V$, and $|V| = n$. The execution of LUBY gives nodes in $V \setminus \{u_1\}$ roughly n times more probability to join the MIS than u_1, resulting in an inequality factor of $\Theta(n)$. Therefore, LUBY is characterized as a (particularly) unfair MIS algorithm with respect to this graph.

3.2.2 RESULTS

In this thesis, we will describe and analyze new fair MIS algorithms for several different graph families. All of our algorithms are randomized and satisfy their stated properties with high probability. Preliminary versions of the results described here (also listed in Table 1.1) appeared in [27, 28].

In more detail, we present an algorithm called FAIRROOTED which calculates fair MISes for rooted trees in $O(\log^* n)$ rounds. We then present an algorithm called FAIRTREE

that computes a fair MIS for unrooted tree and completes in $O(\log n)$ rounds. Note that FAIRROOTED needs less time to calculate MIS fairly than FAIRTREE because nodes have more topological information in rooted trees than unrooted trees. For bipartite graphs, we present an algorithm called FAIRBIPART which constructs a fair MIS in $O(\log^2 n)$ rounds. Moreover, for a k-colorable graph that can be colored by k colors in $f(n, k)$ rounds, we present an algorithm called COLORMIS that computes an MIS in $O(f(n, k) + \log^2 n)$ rounds, with an inequality factor in $O(k)$ (since planar graphs are constant colorable, the inequality factor of COLORMIS will be bounded by a constant and this algorithm is therefore fair for planar graphs). We prove, however, there is no MIS algorithm that can achieve fairness for all graphs by identifying a lower bound graph in which any MIS algorithm will result in an inequality factor at least $\Omega(n)$. These results will be expanded with more detail in the next few sections.

3.2.3 FAIR MIS ALGORITHM FOR ROOTED TREES

Our study of fair MIS algorithms starts with an easy case where G is a rooted tree (recall that a rooted tree is a tree with a selected root node, and each non-root node has a pointer pointing to its parent node). We will describe and analyze a $O(\log^* n)$-time algorithm FAIRROOTED with an inequality factor upper bounded by 4, indicating its fairness. This straightforward result provides a nice primer for our later results and highlights the general strategy we deploy throughout this section: build an initial independent set with strong fairness guarantees, then refine it (using a potentially unfair MIS algorithm on the remaining uncovered nodes) to guarantee maximality.

ALGORITHM

The algorithm FAIRROOTED runs in a rooted tree $T = (V, E)$. It is trivial to extend this algorithm to operate on a forest of rooted trees.

FAIRROOTED ($\forall v \in V$)

Stage 1:

choose $v.tag$ from $\{0, 1\}$ uniformly at random
if v is not the root **then**
  read $u.tag$ from parent node u
else
  randomly generate a virtual parent node tag $u.tag$ from $\{0, 1\}$
if $v.tag = 0$ and $u.tag = 1$ **then** v joins $\mathcal{I}$

Stage 2:

if $v \in \mathcal{I}$, **then**
  output "in MIS" and terminate
else if $N_T(v) \cap \mathcal{I} \neq \emptyset$ **then**
  output "not in MIS" and terminate
else run (ignoring already terminated nodes) Cole and Vishkin's MIS algorithm [17]

Figure 3.1: The FAIRROOTED algorithm run by node $v \in V$. The algorithm runs in two stages.

The algorithm proceeds in two stages, as shown by pseudocode in Figure 3.1. During the first stage, which consists of a single round of communication, each node tags itself with a single bit chosen uniformly at random. We associate with the root a virtual sentinel node v_0 to act as its parent, and have the root also choose the tag of its (virtual) parent. Each node then shares its tag with its children and compares its own tag to that of its parent. Any node with a tag of 0 whose parent has a tag of 1 enters the set $\mathcal{I}$.

By construction, the set $\mathcal{I}$ is an independent set, but it may not be maximal. It is not hard to show that the first stage alone guarantees that each vertex enters $\mathcal{I}$ with constant probability, which implies fairness. In the second stage, each node communicates once with all its neighbors, and any node that is in $\mathcal{I}$, or learns that it is covered by a node in $\mathcal{I}$, terminates the algorithm arriving at its final state. Those nodes that are not yet covered

continue by participating in Cole and Vishkin's MIS algorithm that terminates in $O(\log^* n)$ rounds deterministically [17].

ANALYSIS

The following theorem proves the desired correctness, time complexity, and fairness results for FAIRROOTED.

Theorem 3.2.4. *Let $\mathcal{R}$ denote the class of rooted trees. When run on any rooted tree $T \in \mathcal{R}$,* FAIRROOTED *generates a correct MIS in $O(\log^* n)$ rounds. Moreover, it guarantees an inequality factor $F_{\text{FAIRROOTED}}(\mathcal{R}) \leq 4$.*

The proof of the theorem follows directly from the three subsequent lemmas, which prove correctness, running time, and fairness, respectively.

Lemma 3.2.5. *When run in any rooted tree $T = (V, E)$,* FAIRROOTED *computes a correct MIS.*

Proof. The proof is straightforward. It follows directly from the definition of the algorithm that in Stage 1, no two neighbors may join $\mathcal{I}$ giving independence, and in Stage 2, maximality is restored. □

Since Stage 1 and 2 use a constant number of rounds plus Cole and Vishkin's algorithm for rooted trees [17], it is straightforward to prove the following lemma. Moreover, Cole and Vishkin's algorithm is deterministic, so the bound holds in the worst case if nodes are given unique IDs in the range from 0 to $n^{\Theta(1)}$. We assume in the following that nodes have unique IDs from appropriate range. If not, the nodes begin by choosing random IDs in this range, and the bound holds with high probability.

Lemma 3.2.6. FAIRROOTED *completes in $O(\log^* n)$ rounds.*

Proof. It follows from the fact that Stage 1 requires only constant number of rounds while Cole and Vishki's MIS algorithm terminates in $O(\log^* n)$ rounds [17]. □

Lemma 3.2.7. *Let $\mathcal{R}$ be the class of rooted trees.* FAIRROOTED *has inequality factor $F_{\text{FAIRROOTED}}(\mathcal{R}) \leq 4$.*

Proof. Consider any node $v \in V$, and let u be its parent. Then in the first stage, we have

$$\begin{aligned} \Pr\{v \in \mathcal{I}\} &= \Pr\{u.tag = 1 \text{ and } v.tag = 0\} \\ &= \Pr\{u.tag = 1\} \cdot \Pr\{v.tag = 0\} \\ &= (1/2)(1/2) = 1/4. \end{aligned}$$

In Stage 2, nodes are only *added* to the MIS, so we conclude that the probability that any node enters the MIS is at least $1/4$. Since the maximum probability is 1, the inequality bound follows. □

3.2.4 FAIR MIS ALGORITHM FOR UNROOTED TREES

We now turn our attention to unrooted trees. It is more complicated to compute MIS fairly in these trees as compared to trees with roots. Intuitively, this makes sense as the parent/child relationship in rooted trees provides some symmetry breaking "for free". Here we describe the FAIRTREE algorithm that computes a fair MIS in $O(\log n)$ rounds with high probability, matching the time complexity of LUBY. Notice, this time complexity is slightly slower than the best known (unfair) MIS algorithm for unrooted trees, which runs in $O(\sqrt{\log n \log\log n})$ rounds [73]. FAIRTREE uses CNTRLFAIRBIPART as subroutine—a centralized protocol that can generate a perfectly fair MIS on unrooted trees (or, more generally, any bipartite graph) in $O(D(T))$ time, where T is the tree in which it is executed, and $D(T)$ is T's diameter. FAIRTREE guarantees that each node has a probability of no greater than $(1-\epsilon)/4$ to be selected in MIS by FAIRTREE, where $\epsilon \leq 1/n$.

ALGORITHM

Figure 3.2 describes FAIRTREE, which includes four stages. In the first stage, the unrooted tree is split into smaller components by "cutting" edges randomly. CNTRLFAIRBIPART is then run on each resulting component. The second stage deals with MIS conflicts created when the algorithm subsequently adds back the cut edges connecting the components. It does so by running CNTRLFAIRBIPART on the subtree induced by MIS nodes selected in the previous stage. The third stage uses CNTRLFAIRBIPART on nodes that remain uncovered by the MIS, and the last stage executes LUBY, a randomized MIS algorithm from [78] that terminates in $O(\log n)$ rounds with high probability, to fix any remaining maximality errors. This additional stage is called only to ensure the correctness of our MIS, and with high probability nodes will not execute this stage.

Below we describe FAIRTREE, and the CNTRLFAIRBIPART subroutine, in more detail.

The CNTRLFAIRBIPART Algorithm. Here we describe and analyze the key CNTRLFAIRBIPART subroutine used multiple times in FAIRTREE. The algorithm takes a single parameter, $\widehat{D}$, which is an estimated upper bound on the diameter of the bipartite graph in which it is executed.[1] The algorithm starts by having each node run a basic flood-based leader election algorithm for $\widehat{D}$ rounds: each node in each round broadcasts the largest ID it has received so far; it accepts the largest ID its seen at the end of $\widehat{D}$ rounds as its leader. After this stage concludes, the leader u (or, potentially multiple leaders if $\widehat{D}$ is an underestimate), selects a bit b_u with uniform randomness. It then initiates a breadth-first search, beginning at itself, terminating after $\widehat{D}$ rounds, even if some nodes have not been reached. The search message includes the current depth of the search (u considers itself at level 0) and b_u. Each node (including u), that learns it is in some level i (i hops away from the leader), joins the

[1]Note that we do not assume advance knowledge of diameter information in our model. The CNTRLFAIRBIPART algorithm described here will be called as a subroutine by our FAIRTREE algorithm, which will be responsible for specifying the $\widehat{D}$ parameter.

FAIRTREE
Stage 1: Cut ($\forall v \in V$)
cooperate with each neighbor $u \in N_T(v)$,   and set $(u, v).cut = 1$ with probability $1/2$
call CNTRLFAIRBIPART with $\widehat{D} = \gamma$,   but ignoring edges with $cut = 1$
if v joined MIS **then** add v to $\mathcal{I}$
Stage 2: Resolve ($\forall v \in \mathcal{I}$)
call CNTRLFAIRBIPART with $\widehat{D} = \gamma$
if v joined MIS **then** keep v in $\mathcal{I}$
else remove v from $\mathcal{I}$
Stage 3: Maximalize ($\forall v$ s.t. $(N_T(v) \cup \{v\}) \cap \mathcal{I} = \emptyset$)
call CNTRLFAIRBIPART with $\widehat{D} = \gamma$
if v joined MIS **then** add v to $\mathcal{I}$
Stage 4: Fix ($\forall v \in V$)
if $v \in \mathcal{I}$ and $N_T(v) \cap \mathcal{I} \neq \emptyset$ **then**   remove v from $\mathcal{I}$
if $v \in \mathcal{I}$ **then** output "in MIS" and terminate
else if $N_G(v) \cap \mathcal{I} \neq \emptyset$ **then**   output "not in MIS" and terminate
else   call LUBY (ignoring already terminated nodes) and mimic output

Figure 3.2: The four stages of FAIRTREE algorithm. In the above, $\gamma = \hat{c} \log n$, for a constant $\hat{c} \geq 1$ that we fix in our analysis. Each stage runs for a fixed number of rounds (potentially dependent on γ). Only the nodes specified by the stage title participate in each stage. All nodes, including those not participating, wait the fixed number of rounds for the stage to terminate before moving to the next stage, ensuring all nodes start each stage during the same round.

MIS if $i + b_u \equiv 0 \pmod 2$. As a special case, if the leader is alone (i.e., has degree 0), it always joins the MIS. It is straightforward to show:

Lemma 3.2.8. *Assume* CNTRLFAIRBIPART *is called by every node in unrooted tree* $T = (V, E)$ *during the same round with the same parameter* $\widehat{D}$*. Let* $I(T)$ *be the nodes that join the MIS. If* $\widehat{D} \geq D(T)$*, then: (a)* $I(T)$ *is a correct MIS for* T*; and (b)* $\forall u \in V$: $P_{\text{CNTRLFAIRBIPART},T}(u) = 1/2$ *if* $|V| > 1$*, and* $P_{\text{CNTRLFAIRBIPART},T}(u) \geq 1/2$ *in general.*

Proof. If $\widehat{D} \geq D(T)$, then the leader election correctly elects a single leader and the breadth-first search reaches all nodes. Because T is a tree, it is easy to see that (a) holds. To show (b), fix some node $v \in T$. Let u be the leader in T. If $|V| > 1$, then depending on the parity of $d_T(u, v)$, one value for b_u will put v in $I(T)$ and one will keep v out of $I(T)$. The fairness follows from the fact that b_u is chosen with uniform probability. If $|V| = 1$, then the node always enters the MIS. □

The FAIRTREE Algorithm. We now describe the FAIRTREE algorithm (see Figure 3.2 for pseudocode). This algorithm consists of four stages. Notice, not all nodes participate in each stage (the participants are specified by the set next to the stage name). The first three stages, however, each run for a fixed number of rounds (the $\Theta(\log n)$ time required by the call to CNTRLFAIRBIPART, plus the constant number of extra rounds needed for local communication, when relevant), so non-participants simply wait that number of rounds before proceeding to the next stage. This ensures all nodes start each of these stages during the same round.

Stage 1 divides the tree into components by cutting edges and then attempts to create a fair MIS $\mathcal{I}$ in each component. If Stage 1 completes, $\mathcal{I}$ covers all nodes, but there may be MIS conflicts between neighbors in distinct components. Stage 2 resolves these conflicts by running CNTRLFAIRBIPART only on "MIS" nodes (those in $\mathcal{I}$), causing some nodes to drop out of $\mathcal{I}$. If both stages complete, then $\mathcal{I}$ is an independent set, albeit not necessarily

maximal. Stage 3 restores maximality by running CNTRLFAIRBIPART on uncovered nodes. The resulting MIS stitches safely with the existing set $\mathcal{I}$.

In the analysis that follows, we will prove that the components executing CNTRLFAIRBIPART in each stage have sufficiently small diameters for this fixed-time algorithm to succeed, with high probability. In this successful case, $\mathcal{I}$ is a valid MIS, and we shall prove that $P_{\text{FAIRTREE},T}(u) \geq 1/4$ for every node u. With low probability, however, we might arrive at Stage 4 with an invalid MIS due to CNTRLFAIRBIPART failing to complete on a large diameter component in previous stages. To correct for this possibility, at the start of Stage 4 we remove all independence violations, then have uncovered nodes run a standard MIS algorithm, in this case, LUBY. The resulting MIS will stitch together with the earlier independent set, but we make no guarantee about its inequality factor. In other words, LUBY is only ever called as a "fallback mode" in the low probability event that one of the previous three stages failed. This ensures that the MIS is always correct, but the join probability decreases by some $\epsilon \leq 1/n$.

ANALYSIS

The following theorem proves the desired inequality factor and time complexity results for FAIRTREE.

Theorem 3.2.9. *For any unrooted tree $T = (V, E)$, the* FAIRTREE *algorithm, when executed in T, constructs a correct MIS such that $P_{\text{FAIRTREE},T}(u) \geq (1-\epsilon)/4$, for every $u \in V$ and some $\epsilon < 1/n$. It terminates in $O(\log n)$ rounds with high probability.*

To establish this theorem, we have three properties to prove: the time complexity, correctness, and inequality factor bound of FAIRTREE. We divide these efforts into the three lemmas below. Moreover, in order to distinguish the value of our set $\mathcal{I}$ at the end of each

stage, we use the notation $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$ to describe $\mathcal{I}$ at the end of Stages 1, 2 and 3, respectively.

Lemma 3.2.10. *Algorithm* FAIRTREE *terminates in* $O(\log n)$ *rounds with high probability.*

Proof. This time complexity follows directly from the fixed-length of Stages 1 to 3 and the $O(\log n)$ bound (*w.h.p.*) on LUBY proved in [78]. □

Lemma 3.2.11. FAIRTREE *generates a correct MIS on unrooted tree* T.

Proof. Stage 4 starts by removing any independence violations, then ensures maximality by running LUBY on the remaining uncovered nodes. □

As seen above, efficiency and correctness are straightforward to prove. Bounding the inequality factor, on the other hand, requires a more involved argument:

Lemma 3.2.12. $P_{\text{FAIRTREE},T}(u) \geq (1-\epsilon)/4$, *for all* $u \in V$ *and some* $\epsilon \leq 1/n$.

Proof. We consider Stages 1–3 *successful* if their calls to CNTRLFAIRBIPART are made with a sufficiently large value for $\widehat{D}$ (i.e., a value at least as large as the largest relevant component).

We will first show that if Stages 1–3 are successful, then no node will call LUBY in Stage 4. To see why this is true, notice that a node calls LUBY only if it ends Stage 3 uncovered by $\mathcal{I}$. Every node that is uncovered at the beginning of Stage 3, however, calls CNTRLFAIRBIPART, and by our assumption that this call is successful, each of these nodes ends the stage covered.

We have just established that if the first three stages are successful then no node will call LUBY. In other words, the MIS defined at the end of Stage 3 will be the final MIS. We next note that under this assumption of successful stages, this final MIS is fair. In more detail, the probability that a given u joins the MIS is at least

$$\Pr\{u \in \mathcal{I}_2\} = \Pr\{u \in \mathcal{I}_1\} \cdot \Pr\{u \in \mathcal{I}_2 | u \in \mathcal{I}_1\}.$$

(Since Stage 3 only increases the join probability, it may be omitted when proving a lower bound.) Lemma 3.2.8 implies Stage 1 yields $\Pr\{u \in \mathcal{I}_1\} \geq 1/2$. Stage 2 runs CNTRL-FAIRBIPART again on the nodes $\mathcal{I}_1$, giving $\Pr\{u \in \mathcal{I}_2 | u \in \mathcal{I}_1\} \geq 1/2$. Combining these gives $\Pr\{u \in \mathcal{I}_2\} \geq 1/4$.

Our final step is to incorporate the event that the first three stages are not successful. If this event occurs, for any u, we can trivially lower bound u's probability of joining at 0. Let ϵ be the probability that at least one of these stages fails. Combined with our result from above, we get that u joins the MIS with probability at least $(1-\epsilon)/4$.

We are left to bound ϵ. Our goal is to show that it is less than $1/n$, which will imply the inequality factor is upper bounded by a value that approaches 4 as n increases. To reach this goal, we argue that there exists a constant c for $\gamma = c \log n + \Theta(1)$ such that in all three calls to CNTRLFAIRBIPART, γ is a sufficiently large estimate of the maximum component diameter. We can then set $\hat{C}$ (in the pseudocode from Figure 3.2) to any constant such that $\hat{c} \log n$ upper bounds $c \log n + \Theta(1)$.

We start by studying the calls in the first two stages. In these stages, a given path of length ℓ is included in a component with probability $2^{-\ell}$ (in Stage 1, the path must have $cut = 0$ for every edge, while in Stage 2, the path must have $cut = 1$ for every edge, where the cut decisions are all independent and uniform). Setting $\gamma = c \log n$ for a sufficiently large constant c, a union bound over the polynomially-many paths in T shows that for each of Stages 1 and 2, $\Pr\{\text{length of any path} \geq \gamma\} \leq 1/(3n)$.

For Stage 3, we note that for a path $u_1, u_2, ..., u_\ell$ of length ℓ consisting of nodes uncovered by $\mathcal{I}_2$, each u_i was never previously in $\mathcal{I}$ (if it was, it would be covered in this stage), and, therefore, each u_i has a neighbor v_i that was in $\mathcal{I}_1$ but not $\mathcal{I}_2$. Furthermore, because T is a tree, for v_i, v_j, $i \neq j$, v_i and v_j are in different components in Stage 2 (they are separated by the path $u_i, \ldots, u_j$) and therefore their outcomes in Stage 2 are independent. Lemma 3.2.8 thus implies $\Pr\{v_i \notin \mathcal{I}_2\} \leq 1/2$ independently for each v_i. The same argu-

ment as for Stages 1 and 2 shows that for sufficiently large γ, the probability that Stage 3 contains a large-diameter component is at most $1/(3n)$. A final union bound over stages gives us the desired $\epsilon < 1/n$. □

3.2.5 FAIR MIS ALGORITHM FOR BIPARTITE GRAPHS

We now consider bipartite graphs. Note that unrooted trees are merely a subset of bipartite graphs that excludes cycles. We describe a fair distributed MIS algorithm for the class of bipartite graphs, FAIRBIPART, that runs in $O(\log^2 n)$ time. This algorithm generalizes the algorithm of Section 3.2.4, but does not subsume it because it is slower by a $\log$-factor. Similar to FAIRTREE, the main idea of FAIRBIPART is to partition the graph into low-diameter subgraphs, find an MIS on the subgraphs using CNTRLFAIRBIPART, and then correct for maximality by running LUBY on the uncovered nodes. Unfortunately, Section 3.2.4's partitioning algorithm does not generalize to bipartite graphs, so a more pliable approach is necessary. Fortunately, such an approach exists in a classic network decomposition result due to Linial and Saks [77], which we leverage to quickly produce a useful low weak-diameter subgraph in $O(\log^2 n)$ rounds.

Below, we describe and analyze the useful subroutine we adopt from [77], then describe and analyze our fair MIS algorithm that makes use of it.

CONSTRUCT_BLOCK ROUTINE

Linial and Saks' network decomposition algorithm [77] uses a key subroutine called "Construct_Block." In this routine, each node v chooses a communication range r_v according to the distribution π, specified as:

$$\pi : \Pr\{r_v = k\} = \begin{cases} p^k(1-p) & \forall k \in 0, 1, \ldots, \gamma - 1 \\ p^\gamma & k = \gamma \end{cases} \tag{3.1}$$

where the parameters p and γ represent a probability and maximum range, respectively. For our purposes, $p = 1/2$ and $\gamma = \Theta(\log n)$ suffice.

Next, each node v broadcasts its ID to all other nodes within radius r_v. After the broadcast completes, node v identifies the node u with the largest ID among messages it has received. It considers u its leader. If the distance between v and u is strictly less than r_u, then v joins u's ***block***. In other words, u's block is the set of nodes that select u as a leader and join a block. Otherwise, the distance between v and its leader is exactly r_u, in which case v is called a ***boundary node***. Boundary nodes do not join any block.

As stated in the following lemma, proved in [77], the Construct_Block routine provides some nice structural guarantees. Notably for correctness, if a node v joins a block, then each of its neighbors is either a boundary node or in the same block.

Lemma 3.2.13. *Construct_Block has the following properties:*
(i) each vertex belongs to a block with the probability at least $p(1 - p^\gamma)^n$*;*
(ii) all connected non-boundary nodes have the same leader. □

Our FAIRBIPART algorithm uses a modified version of Construct_Block in its first stage. This modified version implements the same basic operation of Construct_Block as described above, but is modified to also run a version of CNTRLFAIRBIPART along with the block construction, resulting in blocks in which nodes have already placed themselves in a fair MIS. This modification of Construct_Block requires $O(\log^2 n)$ rounds. Though this is slower by a $O(\log n)$ factor than our FAIRTREE algorithm for unrooted trees, the structural properties on the blocks (as described by Lemma 3.2.13) are stronger than what we get from our FAIRTREE partitioning. In particular, the blocks we obtain here are separated by non-block boundary nodes (follows from ii), allowing us to avoid independence conflicts between neighboring nodes in different blocks.

ALGORITHM

Figure 3.3 contains pseudocode describing FAIRBIPART, our fair distributed MIS algorithm for bipartite graphs. Roughly speaking, the goal of Stage 1 is to run a modified version of Construct_Block which creates blocks while simultaneously implementing the logic of CNTRLFAIRBIPART to create a fair MIS in each block. The reason it is necessary to run these two operations simultaneously (instead of one after another) is that each block created by Construct_Block has only weak diameter guarantees in the sense that the distance in G between any two nodes in the same block is small (by construction), but the distance between those nodes in the subgraph induced by the block nodes may be arbitrarily large. To compensate for this reality, Stage 1 simulates CNTRLFAIRBIPART along with the execution of Construct_Block by sending the extra random bit selected by the leader along with each message.

For concreteness, Figure 3.3 provides full pseudocode for Stage 1. There are several variants for Construct_Block described in [77]. Here we adopt one with bounded message sizes, allowing $O(\log n)$ bits per edge per round of communication (as required by our model). More precisely, Stage 1 operates as follows. Each node v initially selects a range or $r_v \leq \gamma = \Theta(\log n)$ according to the distribution π described above. To simulate CNTRLFAIRBIPART, each node also selects a random bit b_v. Each node v constructs "leader tables" $v.L[0\,..\,\gamma]$ and $v.B[0\,..\,\gamma]$, where

$$v.L[i] = \text{maximum ID } v \text{ has seen with } i \text{ range remaining,}$$

$$v.B[i] = \begin{cases} b_u & d(u,v) \text{ is even, where } u = v.L[i] \\ \neg b_u & d(u,v) \text{ is odd, where } u = v.L[i] \end{cases}$$

This table is initialized with $v.L[r_v] = v$ and $v.B[r_v] = b_v$, and all other entries initialized to null values.

FAIRBIPART ($\forall v \in V$)

Stage 1: modified Construct_Block

pick r_v according to distribution π
choose b_v from $\{0, 1\}$ uniformly at random
create tables $v.L[0\,..\,\gamma]$ and $v.B[0\,..\,\gamma]$
initialize $v.L[r_v] \leftarrow v$ and $v.B[r_v] \leftarrow b_v$

repeat γ times
  broadcast leader table $v.L$ and $v.B$
  on receiving leader tables L^*, B^*:
    for all $i \in \{0, 1, 2, \ldots, \gamma - 1\}$ **do**
      if $v.L[i] < L^*[i+1]$ **then**
        $v.L[i] \leftarrow L^*[i+1]$
        $v.B[i] \leftarrow \neg B^*[i+1]$

let $leader\_ID = \max_i v.L[i]$ and $j = \text{argmax}_i v.L[i]$
if $leader\_ID \neq v.L[i]$ for any $i > 0$ **then**
  v is a boundary node and does not join a block
else v joins $leader\_ID$'s block
  if $v.B[j] = 1$ **then** v joins $\mathcal{I}$

Stage 2: MIS

if $v \in \mathcal{I}$ **then**
  output "in MIS" and terminate
else if $N(v) \cap \mathcal{I} \neq \emptyset$ **then**
  output "not in MIS" and terminate"
else
  call LUBY (ignoring nodes that are already terminated) and mimic output

Figure 3.3: The algorithm of FAIRBIPART. Here distribution π is defined in Equation 3.1, and $\gamma = \hat{c} \log n$, for a constant $\hat{c} \geq 1$ that we fix in our analysis.

The execution of Stage 1 is then divided into γ superrounds. In each superround, a node v sends its full leader tables to its neighbors—since a leader table has γ entries, this can be simulated in $O(\gamma)$ rounds. On receiving a leader table, v decrements the range of all received entries by 1 and updates the appropriate entries in its own leader tables to reflect the maximum ID seen.

After completing the γ superrounds of communication, v selects a leader by looking at the maximum ID stored in its leader table $v.L$; i.e., $\max_i v.L[i]$. If that leader's ID occurs only in $v.L[0]$ then v becomes a boundary vertex. Otherwise, it joins a block. To finish the simulation of CNTRLFAIRBIPART, if v joins a block, and if the corresponding bit in $v.B$ is 1, then v joins the MIS $\mathcal{I}$.

At the start of Stage 2, $\mathcal{I}$ is an independent set but may not be maximal. All nodes that are already covered terminate. Any remaining nodes execute LUBY to restore maximality.

ANALYSIS

The following theorem proves the desired fairness and time complexity results for FAIRBIPART. In the following, assume we fix $\gamma = 2 \lg n$ and $p = 1/2$.

Theorem 3.2.14. *Let $\mathcal{B}$ denote the class of bipartite graphs. When run on any bipartite graph $G \in \mathcal{B}$,* FAIRBIPART *generates a correct MIS in $O(\log^2 n)$ rounds, with high probability. Moreover, it guarantees inequality factor $F_{\text{FAIRBIPART},\mathcal{B}} \leq 8$.*

Proof of the theorem follows directly from the subsequent three lemmas.

Lemma 3.2.15. FAIRBIPART *computes a correct MIS.*

Proof. This proof amounts to showing that $\mathcal{I}$ produced at the end of Stage 1 is an independent set. If $\mathcal{I}$ is an independent set, then the fact that Stage 2 yields an MIS follows from the same style of argument as in Lemma 3.2.5.

To prove that $\mathcal{I}$ is an independent set, we begin by applying Lemma 3.2.13. Fix some v with leader u at the end of the Construct_Block logic. All of v's neighbors are either in u's block, or they are boundary nodes. Boundary nodes do not join $\mathcal{I}$, so it is sufficient to show that nodes in the same block that join $\mathcal{I}$ are independent. Below we prove that this follows from the correctness of the CNTRLFAIRBIPART logic integrated in this routine.

We begin by stating an important property of bipartite graphs, which we then use to argue that no neighbors in the same block read the same $v.B[j]$ value at the end of Stage 1. Consider any nodes $u, v \in G$. We say that these nodes have *even distance* if any path between them has even length, and *odd distance* if any path between them has odd length. Since G is bipartite, the notion of even/odd distance is well defined—all paths between a particular pair of vertices have the same parity.

Consider any node v, and let $u = v.L[i]$ be the ith entry in its leader table. A simple inductive argument over rounds of Stage 1 shows that $v.B[i] = b_u$ if and only if u and v have even distance (recall that the algorithm negates this value with each hop). It follows that all of v's neighbors with u in their leader table observe the value $\neg b_u$. Hence v joins $\mathcal{I}$ only if its neighbors do not. □

Lemma 3.2.16. *Algorithm* FAIRBIPART *terminates in* $O(\log^2 n)$ *rounds with high probability.*

Proof. The communication in Stage 1 consists of γ superrounds, which each includes γ rounds. Because we fixed $\gamma = \Theta(\log n)$, we get $O(\log^2 n)$ total rounds. In addition, Stage 2 comprises one round of communication to determine if $N(v) \cap \mathcal{I} = \emptyset$, and we then require $O(\log n)$ rounds (with high probability) for LUBY. □

Lemma 3.2.17. *If* $G \in \mathcal{B}$ *is a bipartite graph, then* $P_{\text{FAIRBIPART},G}(u) \geq 1/8$ *for all* $u \in V$.

Proof. A node v joins $\mathcal{I}$ at the end of Stage 1 if 1) it joins a block, and 2) the bit corresponding to its leader in $v.B$ is 1. Notice, these two events are independent. It is easy to see that 2 occurs with probability $1/2$.

We now turn our attention to the probability of 1. By Lemma 3.2.13, we know that each node joins a block with probability at least $p(1-p^{\gamma})^n$. As specified above, we fixed $\gamma = 2\lg n$ and $p = 1/2$, which yield a block join probability of $(1/2)(1-1/n^2)^n$. Notice that this function is monotonically increasing (approaching $\sqrt{1/e}$ as $n \to \infty$). By assuming $n \geq 2$, we lower bound the probability as: $(1/2)(1-1/4)^2 > 1/4$. (The $n = 1$ case can be handled separately.)

Multiplying the probabilities of events 1 and 2 give us a result $\geq (1/4)(1/2) = (1/8)$, completing the proof. □

Notice, that in the above calculation we can drive the inequality bound arbitrarily close to 4 by replacing $2\lg n$ with $c\lg n$ for increasing values of c (which pushes the probability joining a block in the above proof toward $1/2$ in the limit as c grows). This increased fairness, however, comes at the expense of time complexity, where γ appears as a multiplicative factor.

3.2.6 "FAIR" MIS ALGORITHM FOR k-COLORABLE GRAPHS

We now consider a more general class of graphs: those that are k-colorable (graphs that can be colored with no more than k colors) for which there exists a distributed algorithm that achieves this k-coloring.

In more detail, we will describe and analyze the COLORMIS algorithm for k-colorable graphs. Fix a distributed k-coloring algorithm $\mathcal{A}$ that completes in $f(n,k)$ rounds on some graph G. COLORMIS uses $\mathcal{A}$ as subroutine, and completes in $O(f(n,k)+\log^2 n)$ rounds with high probability, when run on G. The corresponding inequality factor is $O(k)$.

An important corollary will be that COLORMIS is fair MIS algorithm (i.e., its inequality factor is bounded by a constant) that terminates in $O(f(n,k) + \log^2 n)$ rounds when run in planar graphs. This follows because the algorithm in [8] colors any planar graph with a constant number of colors in $O(\log n)$ rounds.

ALGORITHM

Our COLORMIS distributed MIS algorithm must be combined with a distributed k-coloring algorithm $\mathcal{A}$. It begins by having nodes execute $\mathcal{A}$.[2] It then has nodes execute a variant of our augmented Construct_Block subroutine described and analyzed in Section 3.2.5. The only change is that we replace the randomly generated bit b_u generated by each node u, with a color c_u, selected from the k colors used by $\mathcal{A}$ with uniform randomness. (If nodes do not know k in advance, then we can add an extra step where the leader in each block counts the colors before randomly choosing one. For concision, in the following we assume knowledge of k.)

Unlike with b_u, the selected value c_u remains unchanged as it propagates through the network. A node v joins the MIS at this point only if it joined a block with leader u and v's color, from executing the distributed coloring algorithm, matches the color c_u randomly chosen by its leader. As usual, we conclude by having all uncovered nodes execute LUBY to fix any remaining maximality mistakes. Because this algorithm is modified version of the algorithm from Section 3.2.5, we omit a standalone pseudocode description.

ANALYSIS

The following theorem bounds the correctness, time complexity and fairness of COLORMIS:

[2] Nodes can execute $\mathcal{A}$ for a fixed number of rounds that describe its high probability termination bound. In the low probability event that a node remains uncolored, it just proceeds to next step uncolored.

Theorem 3.2.18. *Fix some distributed k-coloring algorithm $\mathcal{A}$ that terminates in $f(n,k)$ rounds in all graphs of size n, with probability at least $1 - 1/n^2$. Let $\mathcal{C}_\mathcal{A}$ be the class of graphs that can be k-colored by $\mathcal{A}$. When run on any $G \in \mathcal{C}_\mathcal{A}$,* ColorMIS *using $\mathcal{A}$ generates a correct MIS in $O(f(n,k) + \log^2 n)$ rounds, with probability at least $1 - 1/n$. Moreover, it guarantees an inequality factor in $O(k)$.*

Proof. The time complexity follows from the running time of $\mathcal{A}$ and the running time of block construction, as established in Section 3.2.5.

To show that the resulting MIS is correct, it is enough to show that there are no independence violations in the set *before* the remaining uncovered nodes run Luby. Notice two nodes in the same block join the set only if they share the block leader's color. And by the correctness of $\mathcal{A}$, no two neighbors share the same color.

Finally, we consider fairness. The analysis of block construction in Section 3.2.5 established that a node joins a block with constant probability. If a node joins a block, the probability that its leader chose its color is $1/k$. Therefore, all nodes join with probability at least $\Omega(1/k)$ yielding the needed $O(k)$ bound on the inequality factor. □

We obtain the following corollary by combining this theorem with coloring algorithms for low-arboricity graphs [8] that produce an $(\lfloor(2+\epsilon)\cdot a(G)\rfloor+1)$-coloring in $O(a(G)\log n)$ time, where $a(G)$ is the "arboricity" of G. and $\epsilon > 0$ is an arbitrarily small parameter. Coupled with the fact that planar graphs have arboricity at most 3 yields the following corollary. Note that this bound actually holds for any constant-arboricity graph.

Corollary 3.2.19. *There exists a fair distributed MIS algorithm for planar graphs that runs in $O(\log^2 n)$ time, with high probability.* □

3.2.7 IMPOSSIBILITY OF FAIR MIS FOR GENERAL GRAPHS

Since we have explored different fair MIS algorithms for different graph families, the next question is to ask whether there exists an MIS algorithm that is fair on every graph G.

Unfortunately, we are able to show that no MIS algorithm can achieve fairness on all graphs. Consider, in particular, the *cone* graph $C = (V, E)$, where $V = \{u_0, u_1, ..., u_{2k}\}$, for some $k \geq 1$, and $E = \{(u_i, u_j) \mid i, j > 0\} \cup \{(u_0, u_i) \mid 0 < i \leq k\}$. That is, C consists of a clique among the nodes u_1 to u_{2k}, as well as an edge from u_0 to every node from u_1 to u_k. We prove below that every MIS algorithm has inequality factor $\Omega(n)$ when run in C. This result eliminates the possibility of any *universally fair* MIS algorithm (i.e., an algorithm that can guarantee bounded inequality in all graphs).

An interesting property of C is that the ratio between the largest and smallest degree is constant. This implies that inequality is not just caused by disparities in density in different regions of a graph (e.g., nodes in sparse regions joining with higher probability than nodes in dense regions), but can also have deeper topological roots. A better classification of exactly which properties unavoidably yield inequality remains an intriguing open question.

Theorem 3.2.20. *Suppose C is a cone graph of size n. Then for every MIS algorithm $\mathcal{A}$, $F_{\mathcal{A}}(C) = \Omega(n)$.*

Proof. For conciseness, let $P(.)$ be a shorthand for the function $P_{\mathcal{A},C}(.)$. We first note that if $P(u_i) = 0$ for any $u_i \in V$, then the inequality factor is trivially in $\Omega(n)$ (it would, in fact, be infinite). Assume moving forward, therefore, that all nodes have a join probability strictly greater than 0. We turn our attention to the relationship between u_0 and the k nodes in $S = \{u_{k+1}, \ldots, u_{2k}\}$. Let $p_S = \sum_{u_i \in S} P(u_i)$. Notice, if a node in S joins the MIS then u_0 must also join the MIS, as the MIS node in S would cover $\{u_1, ..., u_k\}$, requiring u_0 to join to preserve maximality. The reverse argument also holds: if u_0 joins then a node in S must also join. It follows that $P(u_0) = p_S$. Because p_S is the sum of $|S| = k$ elements, there

must be some $u^* \in S$ such that $P(u^*) \leq p_S/k$. Pulling together these pieces, it follows that

$$F_{\mathcal{A}}(C) \geq \frac{P(u_0)}{P(u^*)} \geq \frac{p_S}{(p_S/k)} = k = \Omega(|V|) = \Omega(n).$$

□

3.3 FAIR VERTEX COLORING ALGORITHMS

A vertex coloring algorithm assigns a color to each vertex such that no neighbors have the same color. Formally, vertex coloring can be defined as follows [104]:

Definition 3.3.1 (Vertex k-coloring)**.** *Suppose $C = \{1, 2, \ldots, k\}$. Mapping $f : V(G) \to C$ is a vertex k-coloring of graph G if for all $i \in C$, $f^{-1}(i)$, the collection of $v \in V$ such that $f(v) = i$, is either an independent set of G or an empty set.*

Graph G is (vertex) k-colorable if the mapping f in Definition 3.3.1 exists for the given k. The chromatic number of graph G, usually denoted by $\chi(G)$, is the minimum value of k such that G is k-colorable. It has been proved that finding $\chi(G)$ is NP-complete [35].

Although finding the chromatic number is difficult, every graph is trivially $(\Delta + 1)$-colorable, where Δ is the maximum degree. Therefore, many existing studies instead make an effort to find an efficient $(\Delta + 1)$-coloring algorithm. The best known $(\Delta + 1)$-coloring algorithm for general graphs requires $O(\log n)$ rounds with high probability [76]. A deterministic algorithm, on the other hand, may need $2^{O(\log n)}$ rounds [86]. Algorithms can be much more efficient with respect to restricted graph types. For example, a graph with a constant chromatic number can complete a $(\Delta + 1)$-coloring in $O(\log^* n)$ rounds [17]. Sparse graphs, with moderate maximum degrees, can be colored by an algorithm with a time complexity of $O(\Delta^2 + \log^* n)$ [38].

3.3.1 FAIRNESS FOR VERTEX COLORING PROBLEM

The definition of fairness for coloring algorithms requires a different style than what we used for the MIS problem. The output of a node v running a coloring algorithm $\mathcal{A}$ is not simply 0 or 1, but a specific color $f_{\mathcal{A}}(v)$ from the color set C. However, by focusing on the fairness of each color individually, a tractable definition is possible. In more detail, suppose G is a k-colorable graph, and $\mathcal{A}$ is a k-coloring algorithm running on G. According to

Definition 3.3.1, $f_{\mathcal{A}}^{-1}(i)$ for $i \in C$ is an independent set that contains all vertices outputting i by executing $\mathcal{A}$. If we fix a color $i \in C$, we might as well regard $\mathcal{A}$ as an independent set algorithm for graph G, and we can define the inequality factor with respect to color i in a similar way to Definition 3.2.2. We define $P_{\mathcal{A},G} : V \times C \to [0,1], (v,i) \mapsto \Pr\{v \in f_{\mathcal{A}}^{-1}(i)\}$. The inequality factor of algorithm $\mathcal{A}$ with respect to color $i \in C$ can be defined as follows:

Definition 3.3.2. *Suppose $C = \{1, 2, \ldots, k\}$ is a set of colors and $\mathcal{A}$ is a k-coloring algorithm on graph G using color palette C. Then we define the inequality factor of algorithm $\mathcal{A}$ on G with respect to color $i \in C$ as*

$$F_{\mathcal{A}}(G,i) = \max_{u,v \in V} \left\{ \frac{P_{\mathcal{A},G}(u,i)}{P_{\mathcal{A},G}(v,i)} \right\}.$$

If there exists $v \in V$ such that $P_{\mathcal{A},G}(v,i) = 0$, then we fix $F_{\mathcal{A}}(G,i) = \infty$.

The definition above immediately yields a definition of the fairness of a vertex coloring algorithm:

Definition 3.3.3. *A k-coloring algorithm $\mathcal{A}$ using color palette C is fair on graph G if there exists a constant $c > 0$ such that for every $i \in C$, $F_{\mathcal{A}}(G,i) \leq c$.*

Global fair coloring. An obvious way to satisfy this definition of fair coloring is to run a coloring algorithm, then agree on an offset, and have all nodes apply that offset to rotate their colors in a coordinate way. The problem with this proposal is its speed. Any non-trivial global agreement will require time in D, the diameter of the graph, which can be large in n. Note that the fair algorithm studied in this section will be much faster as it requires only $O(\log^2 n)$ time, much faster than the time required for any global coordinate.

3.3.2 AN UNFAIR $(\Delta + 1)$-COLORING ALGORITHM

Here we describe a $(\Delta + 1)$-coloring algorithm called FASTCOLOR. This algorithm does not offer any fairness guarantees. We will use it as a subroutine of our fair coloring algorithm. We emphasize that we did not invent this algorithm. It is a strategy that exists as algorithmic folklore. For the sake of completeness, however, we describe it and provide a new performance analysis.

As shown in Figure 3.4, algorithm FASTCOLOR takes the palette of colors available to v, Π_v, as its input. Node v first randomly select a color f_v from this palette and then broadcasts it to its neighbors. If no neighbor chooses the same color, v will keep color f_v and broadcast it again to its neighbors and becomes colored when the algorithm finally returns f_v. Otherwise, if v receives a color f_u from its neighbor in this round, it removes f_u from its palette and start a new iteration of its main loop. Node v terminates the loop once it keeps the color that no other neighbor picks (and the loop is also terminated when $|\Pi_v| = 1$ because v has no other uncolored neighbor node in this case and it will always keep the only one color left).

We now show that under reasonable conditions FASTCOLOR terminates quickly with a correct coloring.

Theorem 3.3.4. *Assume every process v is passed a color pallete that is strictly larger than v's neighborhood. Then* FASTCOLOR *terminates with a correct coloring in $O(\log n)$ rounds with probability at least $1 - \frac{1}{n^c}$ for any constant $c \geq 1$.*

The proof of this theorem requires the following lemma:

Lemma 3.3.5. *In each iteration of* FASTCOLOR*, each uncolored node that has not yet been colored will become colored with probability at least $1/64$.*

Proof. For a fixed iteration and a fixed uncolored node v, let $\Pi_v = \{c_1, c_2, \ldots, c_k\}$ be the colors that remain free for v to choose at the start of this iteration ($k \geq 2$). Let δ be the size

FASTCOLOR (input Π_v: the palette of colors available to node v)

repeat
  f_v is a color selected from Π_v with uniform randomness
  $is\_free \leftarrow$ TRUE
  broadcast $(f_v, pick)$ to neighbor nodes in $N(v)$
  $R \leftarrow$ **receive**
  foreach $(f_u, pick) \in R$:
    if $f_u = f_v$ **then**
      $is\_free \leftarrow$ FALSE

  if $is\_free =$ TRUE **then**
    broadcast $(f_v, decide)$ to neighbor nodes in $N(v)$
  $R \leftarrow$ **receive**
  foreach $(f_u, decide) \in R$:
    $\Pi_v \leftarrow \Pi_v \setminus \{f_u\}$
until $is\_free =$ TRUE or $|\Pi_v| = 1$
if $|\Pi_v| = 1$ **then**
  $f_v \leftarrow$ the only element in Π_v
return f_v

Figure 3.4: The algorithm of FASTCOLOR for process v with initial color palette Π_v.

of Π_v at the beginning of the first iteration. Recall, by assumption δ is strictly larger than v's neighborhood size.

We first claim that the number of uncolored neighbor nodes is strictly smaller than k. Assume for contradiction that at least k neighbors remain uncolored. Note that $\delta - k$ colors have been chosen by v's neighbors till the beginning of this iteration. It follows that the number of neighbor nodes that have already been colored will be at least $\delta - k$. Then the total number of v's neighbor nodes is at least $\delta - k + k = \delta$, which contradicts our assumption that δ is strictly larger than v's neighbor size.

We then sort Π_v into $\{c_{(1)}, c_{(2)}, \ldots, c_{(k)}\}$, where $P_{(1)} \leq P_{(2)} \leq \ldots \leq P_{(k)}$ and $P_{(i)} = \sum_{\substack{u \in N(v) \\ u \text{ is uncolored}}} P_{u,(i)}$, and $P_{u,(i)}$ is the probability that node u randomly picks color $c_{(i)}$ from

Π_u. Note that $P_{v,(i)} \leq \frac{1}{2}$ for any $i \in [1,k]$ since v will exit the loop and no random selection is needed when Π_v only has one color available. Since we have proved that v has less than k uncolored neighbors, we will have

$$\sum_{i=1}^{k} P_{u,(i)} = \sum_{c_i \in \Pi_u \cap \Pi_v} P_{u,(i)} \leq \sum_{c_i \in \Pi_u} P_{u,(i)} \leq 1.$$

It follows:

$$\sum_{i=1}^{k} P_{(i)} = \sum_{i=1}^{k} \sum_{\substack{u \in N(v) \\ u \text{ is uncolored}}} P_{u,(i)} = \sum_{\substack{u \in N(v) \\ u \text{ is uncolored}}} \sum_{i=1}^{k} P_{u,(i)} \leq \sum_{\substack{u \in N(v) \\ u \text{ is uncolored}}} 1 < k.$$

If we only consider colors $c_{(1)}, c_{(2)}, \ldots, c_{(\lfloor k/2 \rfloor)}$, we have $P_{(\lfloor k/2 \rfloor)} \leq 2$ because otherwise we will get $\sum_{i=\lfloor k/2 \rfloor + 1}^{k} P_{(i)} \geq k$. Consequently, for any $i \in [1, \lfloor k/2 \rfloor]$,

$$\begin{aligned}
& \Pr\{\text{No neighbor of } v \text{ picks color } c_{(i)}\} \\
= & \prod_{\substack{u \in N(v) \\ u \text{ is uncolored}}} (1 - P_{u,(i)}) \geq \prod_{\substack{u \in N(v) \\ u \text{ is uncolored}}} 4^{-P_{u,(i)}} \\
= & \; 4^{-\sum_{\substack{u \in N(v) \\ u \text{ is uncolored}}} P_{u,(i)}} = 4^{-P_{(i)}} \\
\geq & \; 4^{-2} = \frac{1}{16}.
\end{aligned}$$

The probability that v becomes colored (by colors $c_{(1)}, c_{(2)}, \ldots, c_{(\lfloor k/2 \rfloor)}$) will be:

$$\begin{aligned}
\Pr\{v \text{ becomes colored}\} & \geq \sum_{i=1}^{\lfloor k/2 \rfloor} \Pr\{\text{no neighbor picks } c_{(i)} | v \text{ picks } c_{(i)}\} \Pr\{v \text{ picks } c_{(i)}\} \\
& = \sum_{i=1}^{\lfloor k/2 \rfloor} \Pr\{\text{no neighbor picks } c_{(i)}\} \Pr\{v \text{ picks } c_{(i)}\} \\
& \geq \left\lfloor \frac{k}{2} \right\rfloor \cdot \frac{1}{16} \cdot \frac{1}{k} \geq \frac{1}{64}.
\end{aligned}$$

□

Proof (of Theorem 3.3.4). The correctness of FASTCOLOR follows directly from the definition of the algorithm and the assumption that each node has a color palette to work with that is larger than its neighborhood size.

According to Lemma 3.3.5, each node is colored with probability at least $1/64$ in each iteration. Then for any constant $c \geq 1$, the probability that node v is still uncolored after $\frac{c+1}{6} \log n$ iterations is no greater than $(1/64)^{\frac{c+1}{6} \log n} = 1/n^{c+1}$. Taking a union bound on the number of nodes, we will get that the probability of having some node uncolored after $\frac{c+1}{6} \log n$ iterations is no greater than $1/n^c$ for any constant $c \geq 1$. □

3.3.3 FAIR $(\Delta + 1)$-COLORING

We now describe a fair $(\Delta + 1)$-coloring algorithm called FAIRBLOCKCOLOR (see Figure 3.5). This algorithm will use FASTCOLOR as a subroutine.

FAIRBLOCKCOLOR includes two stages: block construction stage and coloring stage. The first stage adopts the idea of Construct_Block [77] to decompose the entire graph into smaller blocks. Instead of having nodes transmitting a random bit (like FAIRBIPART) or a color (like COLORMIS) during block construction, FAIRBLOCKCOLOR has each node propagate a randomly selected offset $T \in [0, \Delta]$. If a node ends up joining a block, it adopts the offset of its block leader. In the second stage, each block node gets colored by running FASTCOLOR with the full palette C and then uses the leader's offset to rotate its color before it claims its color. Boundary nodes, however, remove the claimed colors of its block neighbors from its own palette and run FASTCOLOR with remaining colors.

We now prove the correctness, time complexity, and fairness of FAIRBLOCKCOLOR.

Theorem 3.3.6. FAIRBLOCKCOLOR *terminates with a correct* $(\Delta + 1)$*-coloring in* $O(\log^2 n)$ *rounds with high probability.*

Proof. Nodes in the same block will rotate their colors in the same way in Stage 2. The rotated colors are still in the range of $\{1, 2, \ldots, \Delta + 1\}$. Because all nodes in the same

FAIRBLOCKCOLOR

Stage 1**:** modified Construct_Block ($\forall v \in V$)

pick r_v according to distribution π
T_v is selected from $[0, \Delta]$ with uniform randomness
create tables $v.L[0\,..\,\gamma]$ and $v.T[0\,..\,\gamma]$
initialize $v.L[r_v] \leftarrow v$ and $v.T[r_v] \leftarrow T_v$

repeat γ times
  broadcast leader table $v.L$ and $v.T$
  on receiving leader tables L^*, T^*:
    for all $i \in \{0, 1, 2, \ldots, \gamma - 1\}$ **do**
      if $v.L[i] < L^*[i+1]$ **then**
        $v.L[i] \leftarrow L^*[i+1]$, $v.T[i] \leftarrow T^*[i+1]$

let $leader\_ID = \max_i v.L[i]$ and $j = \text{argmax}_i v.L[i]$
if $leader\_ID \neq v.L[i]$ for any $i > 0$ **then**
  v is a boundary node and does not join a block
else
  v joins $leader\_ID$'s block
  $T_v \leftarrow v.T[j]$

Stage 2**:** coloring ($\forall v \in V$)

$C \leftarrow \{1, 2, \ldots, \Delta + 1\}$
$\Pi_v \leftarrow C$

if v joins a block **then**
  $f_v \leftarrow$ FASTCOLOR(C)
  $f_v \leftarrow ((f_v + T_v) \bmod (\Delta + 1)) + 1$
  broadcast $(f_v, color)$ to its neighbors and output f_v
else
  $R \leftarrow$ **receive** $\Theta(\log n)$ rounds
  foreach $(f_u, color) \in R$:
    $\Pi_v \leftarrow \Pi_v \setminus \{f_u\}$
  select a color f_v by running FASTCOLOR(Π_v)
  output f_v

Figure 3.5: The algorithm of FAIRBLOCKCOLOR. The distribution π used for selecting broadcasting range defined in Section 3.2.5. As in Section 3.2.5, we require that the block size $\gamma = \Theta(\log n)$.

block apply the same rotation, if their coloring is correct before the rotation, it remains correct after the rotation. Since blocks are not adjacent, these rotations cannot create coloring conflicts between different blocks with different rotations.

Note that we start with $(\Delta + 1)$ colors, so after a boundary node eliminated existing colors in its neighborhood, the number of remaining colors is still larger than its neighborhood. Thus, the correctness of Stage 2 is guaranteed by Theorem 3.3.4, and it terminates within $O(\log n)$ rounds with probability at least $1 - \frac{1}{n^c}$ for constant $c \leq 1$.

The time complexity of FAIRBLOCKCOLOR is dominated by the Construct_Block logic which requires $O(\log^2 n)$ rounds. Then FAIRBLOCKCOLOR terminates in $O(\log^2 n)$ rounds with high probability. □

Theorem 3.3.7. FAIRBLOCKCOLOR *is fair.*

Proof. Fix a set of colors $C = \{1, 2, \ldots, \Delta+1\}$. Then we are going to prove that for all $i \in C$, $F_{\text{FAIRBLOCKCOLOR}}(G, i) \leq 4$. If we fix $\gamma = 2 \lg n$ and $p = 1/2$ in distribution π for Stage 1, we will see that each node has a probability more than $1/4$ to join a block (as it is proved in Lemma 3.2.17). Once a block node v selects a color, it then rotates this color according to the block offset which is chosen independently from v's coloring decision. It follows that $P_{\text{FAIRBLOCKCOLOR},G}(v, i) = \frac{1}{\Delta+1}$ for all $i \in C$, and therefore $F_{\text{FAIRBLOCKCOLOR}}(G, i) \leq 4$ for all $i \in C$. □

CHAPTER 4

FAIR WIRELESS COMMUNICATION

In this chapter we study the multicast problem in a wireless communication model. We will seek new solutions that fairly make use of the bandwidth available to receivers. The results presented here detail and extend our preliminary work on this topic [37].

Note that in the study of graph algorithms, we often make the simple assumption that every communication link is reliable. When studying wireless communication, this assumption is no longer valid. In wireless networks, we must take the quality of communication links into account, and a message transmitter needs to specify one transmission speed to send a message, also called *bitrate*, from a set of available speeds. The message will not be delivered if the specified transmission speed exceeds the fastest speed supported by the link. A *rate selection* strategy is therefore necessary to complete the communication task in an efficient way.

In the context of unicast, where each transmission is designated to one single receiver in the network, rate selection strategies are mostly based on feedback information about the fate of packet transmissions [11, 49, 60, 74, 87, 89, 103]. In the context of multicast, however, one information source may need to serve a group of different receivers with different link qualities. This setting makes the problem more challenging because the source cannot easily receive feedback from many receivers all at once. This "blindness" (to packet outcomes) is fundamental in wireless multicast communication. Existing wireless multicast rate selection strategies tend to fall back on slow transmission speeds that are reliable for all links. This strategy, however, is unfair for receivers with higher link quality, since packets

could have arrived at a faster rate. In this chapter, we seek to identify blind rate selection strategies for multicast that are more fair for every receiver, where we define the fairness of such algorithm for a given receiver to be the competitive ratio of its average latency to that of the latency it would experience if packets were sent at the fastest rate its link can support. The goal is to choose rates in such a way that all receivers have reasonably small competitive ratios.

In more detail, suppose L is the greatest packet latency (i.e., the longest time to send a packet) induced by the slowest bitrate supported by the system. Our study will show that a competitive ratio of $O(\log L)$ can be achieved in a network with static links, where the quality of all links does not change during the execution. That is, each receiver receives multicast packets at an average latency only a factor of $O(\log L)$ slower than its smallest achievable latency. We also prove that this competitive ratio is near optimal by providing a lower bound competitive ratio of $\Omega(\log L/\log\log L)$ for rate selection algorithms running on networks with static links. With fading links, however, where the quality of links can vary from round to round, we show that it is impossible to achieve fairness by showing a lower bound competitive ratio of $\Omega(L)$ for deterministic rate selection algorithms and $\Omega(\sqrt{L})$ for randomized rate selection algorithms.

4.1 MODEL

We model wireless devices broadcasting in a synchronous radio network with various link qualities and transmission rates (which are typically called bitrates). The network is modeled as a directed graph $D = (V, E)$, where nodes in V represent wireless devices and edges in E refer to links between devices. We fix a single node $s \in V$ as the information source. The main topology considered is called a single hop network, where the source s has directed edges to all other nodes (i.e., the receivers). If s does not have a direct link to a certain receiver $t \in V$, the transmission is still feasible if there exist a directed path from s to t, and the corresponding network is called a multihop network.

The quality of a link $e \in E$ in round r is captured by assigning a weight function of minimum latency, or the number of rounds required by applying the fastest acceptable bitrate for the link, denoted by $C(r, e)$. Note that the term of latency is typically specified in rounds per packet, and smaller weight function values correspond to higher link qualities. We say that links are static if the weight function of the links does not change, i.e., $C(r, e) = C(r', e)$ for all r, r' and e, and otherwise the links are said to be fading. Assume that all latencies are integers in $[L] = \{1, 2, \ldots, L\}$, where L (> 4) is the slowest transmission latency. For the sake of convenience and notation concision, we also assume that L is a power of 2.

Communication with neighbors in D is implemented with local broadcast. If a node $s \in V$ broadcasts a message in round r with latency $\ell \in [L]$ (this is equivalent to choosing the bitrate ℓ^{-1}), the transmission requires ℓ rounds to complete. The transmitter must wait until the transmission completes before it can start a new transmission. Each neighbor of the transmitter receives the message if the attempted broadcast latency remains at least as large as the minimal acceptable latency for the link throughout the transmission. Formally, a transmission with latency ℓ starting in round r succeeds at a neighbor w of the sender s if

and only if for all $\forall i \in [0, \ell - 1] : C(r + i, (s, w)) \leq \ell$. Otherwise the receiver is not able to receive the message.

We do not make the assumption of a feedback mechanism (e.g., link layer ACKs) for the sender to learn the fate of its transmissions, as in the multicast setting messages are often delivered to multiple receivers simultaneously (the resulting ACKs would collide). We also assume nodes are provided no advance information about the network size or link weights.

4.2 PROBLEM

We study algorithms that multicast a sequence of messages from the designated source to all the receivers in the network. We call such solutions rate selection algorithms as their core behavior is to select transmission rates for the messages it must multicast. We call a rate selection algorithm blind if it receives no information about the fate of its packets. Here we will focus our attention on blind rate selection algorithms.

In more detail, fix a directed network topology graph $D = (V, E)$. Fix a single node $s \in V$ to the information source, also called the sender. We assume D contains a path from s to every other node in V. The sender is provided a sequence of unique packets to multicast to the network. The task of the sender is to deliver these packets to all receivers. That is, for each packet in this sequence, every receiver must eventually successfully receive that packet.

We will focus in particular on the fairness of a given rate selection algorithm. We capture fairness by the competitive ratio of the average latency of packets at a given receiver to the optimal average latency. In more detail, for a given receiver $v \in V$ and duration $T \geq 1$, the average latency of v through T rounds of a given execution of a rate selection algorithm is calculated as T/N_v^T, where N_v^T is the number of unique packets received by v during the first T rounds. Similarly, we define OPT_v^T to be the optimal (smallest) possible average latency of v through T rounds given the definition of the network and its weight function (we sometimes simplify this to OPT where the parameters are clear from context). Notice, as detailed in the sections that follow, the definition of "optimal" becomes somewhat tricky when studying multihop networks. The goal for a fair rate selection algorithm is to minimize the competitive ratio between optimal and achieved average latency. Formally:

Definition 4.2.1. *Fix some function $f : \mathbb{N}^* \to \mathbb{R}$. A deterministic (randomized) blind rate selection algorithm $\mathcal{A}$ is $f(L)$-competitive with respected to network family $\mathcal{D}$ and static/fading links, if the algorithm correctly delivers every packet to every receiver and there exists some $T_0 \geq 1$ such that for every $D \in \mathcal{D}$ and static/fading weight function C, for every receiver v and every duration $T \geq T_0$: the (expected) average latency of v through T rounds of an execution of $\mathcal{A}$ on D with C, is no greater than $f(L) \cdot OPT_v^T$.*

4.3 SINGLE HOP NETWORK WITH STATIC LINKS

We begin by considering single hop networks with static links. That is, networks in which there is a link from the single source s to every receiver, and the weight function C on each link does not change during the execution of rate selection algorithms. In a slight abuse of notation, in the following, for a given weight function C, we use $C(u, v)$ to describe the static weight on the edge $e = (u, v)$.

We start by describing and analyzing a randomized rate selection algorithm called RANDSELECT with a competitive ratio of $O(\log L)$. We then describe and analyze a deterministic algorithm called BCSSELECT, and show it has the same competitive ratio. There are two motivations for this deterministic solution. The first is that rate selection algorithms are often implemented at low layers of the network stack where efficient access to randomness is difficult. Second, the multihop algorithm studied later in this thesis uses this deterministic algorithm as a key building block, and analyzing randomized strategies over multiple hops can be rather difficult. We conclude by proving a lower bound of $\Omega(\log L / \log \log L)$-competitiveness for all blind rate selection algorithms on single hop network with static links. This bound establishes our algorithms are near optimal.

4.3.1 A RANDOMIZED RATE SELECTION ALGORITHM

The most straightforward rate selection strategy is to randomly guess an available latency with uniform randomness. However, this strategy is not necessarily efficient since the resulting competitive ratio might be linear with respect to L. We solve this problem by weighting the selection of a weight so that slower rates are selected less often. In more detail, we will describe and analyze a randomized rate selection algorithm called RANDSELECT that is $O(\log L)$-competitive.

RANDSELECT (for sender s)

Initialization:
  for $j \leftarrow 1$ **to** $\log L$ **do**
    $Q_j \leftarrow Q_0$
    $nextpacket_j \leftarrow p_1$

Transmission:
  do
    select j according to distribution π
    send $nextpacket_j$ with latency $\ell \leftarrow 2^j$
    $pop(Q_j)$
    $nextpacket_j \leftarrow peek(Q_j)$
  while TRUE

Figure 4.1: The RANDSELECT algorithm.

ALGORITHM

Define $\mathcal{I} = [\log L]$ as the index set. The intuition of RANDSELECT is simple. The initial packet queue Q_0 is copied into $\log L$ copies, one for each index $j \in \mathcal{I}$ denoted by Q_j. In every transmission, the source s randomly selects an index j according to a random distribution π defined as follows:

$$\pi(x) : \Pr\{j = x\} = \begin{cases} 2^{-x} & x \in [1, \log L - 1] \\ 2/L & x = \log L \end{cases}.$$

Then the packet at the head of queue Q_j will be popped and broadcast with latency 2^j. This strategy overcomes the issue of packet loss, since every packet will be sent at the slowest latency eventually.

ANALYSIS

Our goal is to prove the $O(\log L)$-competitiveness of RANDSELECT:

Theorem 4.3.1. *The* RANDSELECT *blind rate selection algorithm is* $O(\log L)$*-competitive with respect to single hop networks and static links.*

Proof. Fix any receiver v. To simplify notation, fix $c = C(s, v)$ and assume $\log c$ is a whole number. Recall that in our definition of competitive ratio we require the ratio only hold after some threshold T_0 of rounds. For our purposes, fix $T_0 = L \log L$. It is clear that the optimal algorithm defined with respect to v is the algorithm that transmits every packet with latency c. Therefore, in every execution of length $T \geq T_0$, the optimal sender delivers T/c packets to v yielding an average latency of c. We must now show that the expected (average) latency of our randomized algorithm in an execution of this length is in $O(c \cdot \log L)$.

In our analysis, we focus only on the expected delay between two consecutive samples of $Q_{\log c}$ (row $\log c$ in the packet table). It follows that we are ignoring the possibility of a given packet being delivered to v earlier from a slower latency queue—a reality which only strengthens our analysis.

We begin be considering the case where $c < L$. Starting from any round in which a transmission completes, we describe the sequence of latencies the algorithm chooses until the next sample of latency c by:

$$\ell_1, \ell_2, \ldots, \ell_\gamma, c$$

where ℓ_α ($\forall \alpha = 1, 2, \ldots, \gamma$) are all random variables defined over distribution π for some $\gamma > 0$. The total rounds needed until the next selection of c in a given execution, denoted by t, is the sum of these latencies (and the expected delay we are aiming at is accordingly given by $E[t]$, the expectation of t):

$$t = \sum_{\alpha=1}^{\gamma} \ell_\alpha + c.$$

According to Wald's Equation [101] and the linearity of expectation, the expected latency between samples of latency c can be calculated as:

$$E[t] = E\left[\sum_{\alpha=1}^{\gamma} \ell_\alpha + c\right] = E[\ell_\alpha]E[\gamma] + E[c] = E[\ell_\alpha]E[\gamma] + c.$$

Specifically, we have

$$E[\ell_\alpha] = \sum_{j=1}^{\log L} 2^j \Pr\{\ell_\alpha = 2^j\} = \log L + 1$$

for all $\alpha = 1, 2, \ldots, \gamma$ because ℓ_α are *idd* random variables from the same distribution π.

Notice that random variable γ refers to the number of samples of latencies other than c before the selector hits c. The distribution of γ is similar to a geometric distribution. In more detail, for a given $\Gamma \geq 0$:

$$\Pr\{\gamma = \Gamma\} = (\Pr\{\ell \neq c\})^\Gamma \cdot \Pr\{\ell = c\} = \frac{1}{c}\left(1 - \frac{1}{c}\right)^\Gamma.$$

Then $E[\gamma] = \sum_{\Gamma=0}^{\infty} \Gamma \cdot \Pr\{\gamma = \Gamma\} = c - 1$.

Combining these results, the expected value of t becomes:

$$E[t] = (\log L + 1)(c - 1) + c = (c - 1)\log L + (2c - 1).$$

By linearity of expectation, the expected average latency for an execution of length $T \geq T_0$ is in $O(T/(c \log L))$ as compared to T/c for OPT. The resulting competitive ratio is $O(\log L)$.

To conclude, we must consider the case where $c = L$. The analysis of expected average latency in this case is the same as the above $c < L$ case with the exception that the probability of selection $\ell = L$ is $2/2^{\log \ell}$ as oppose to the $1/2^{\log \ell}$ used for smaller ℓ values. The impact of this extra factor of 2, however, has no effect on the above asymptotic analysis and yields the same $O(\log L)$ competitive ratio. □

A straightforward practical optimization would be to remove a packet from fast latency queues in the case that it is sent first by a slower latency. Note that this optimization does not affect the asymptotic analysis.

4.3.2 A Deterministic Rate Selection Algorithm

We now describe a deterministic blind rate selection algorithm called BCSSelect. We explore deterministic rate selection strategies as they are arguably more feasible for implementation on lower layers of the network stack, also because they simplify our multihop solutions. In fact, BCSSelect can be regarded as the derandomized version of RandSelect, since the only difference between these two algorithms is how indices are selected. We will argue that BCSSelect is $O(\log L)$-competitive, the same competitive ratio as RandSelect.

Algorithm

Our main strategy for derandomizing RandSelect is to leverage a useful object from number theory called the *binary carry sequence* (BCS) [5]. This BCS is defined such that its k^{th} term is the first position with a 1 bit in the binary representation of k.

To use this sequence for our algorithms, we use the deterministic *schedule* function, which takes as input a natural number k and returns the k^{th} value of the BCS. Formally: for $k \in \mathbb{N}^* : schedule(k) = \max\{\alpha \in \mathbb{N} : 2^{\alpha-1}|k\}$. The output of *schedule*, for example, produces the sequence: 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1... Before proceeding to our algorithm description (which uses the *schedule* output to sample queues), we first state some useful facts about $schedule$ (first identified in [18] and reworded here):

Lemma 4.3.2. *Each value $j \in \mathbb{N}$ is selected every 2^j iterations.*

BCSSELECT (for source s)

Initialization:

$k \leftarrow 1$

for $j \leftarrow 1$ **to** $\log L$ **do**

$Q_j \leftarrow Q_0$

$nextpacket_j \leftarrow p_1$

Transmission:

do

$j \leftarrow schedule(k)$

send $(nextpacket_j, k)$ with latency $\ell = 2^j$

$pop(Q_j)$

$nextpacket_j \leftarrow peek(Q_j)$

$k \leftarrow$ UPDATE(k)

while TRUE

UPDATE(k)

if $k = L/2$ **then return** 1

else return $k + 1$

Figure 4.2: The BCSSELECT algorithm.

Lemma 4.3.3. *If $s = schedule(k)$, then:*

i) $\forall t < s$, $\exists r \in [k - 2^{s-2}, k)$ *such that* $t = schedule(r)$*;*

ii) $\forall t < s$, $\exists r \in (k, k + 2^{s-2}]$ *such that* $t = schedule(r)$.

The BCSSELECT algorithm (described in Figure 4.2), applies the $schedule$ function as a subroutine to sample latencies from a (bounded) binary carry sequence—which ensures, as with the random distribution from before, that all latencies are sampled, but small latencies are sampled more often than their slower counterparts. In more detail, the algorithm uses *schedule* to sample the BCS until the first time latency L is sampled, at which point it restarts the sequence. As a result, the sequence of latencies sampled by *schedule* can be

seen as repeating the same bounded BCS block with $L/2$ terms.[1] In addition, the algorithm has the source send the current BCS index k along with the packet, as this will prove useful in the multihop version of the algorithm we study later.

ANALYSIS

We now argue that BCSSELECT is $O(\log L)$-competitive, the same competitive ratio as RANDSELECT.

Theorem 4.3.4. *The* BCSSELECT *blind rate selection algorithm is* $O(\log L)$*-competitive with respect to single hop networks and static links.*

Proof. Fix receiver v with $C(s, v) = c$, where $\log c \in J = [1, \log L]$. Fix $T_0 = L \log L$ as well and suppose the execution runs for $T \geq T_0$ rounds. By the same token, we have $OPT = c$ because the optimal solution runs with the sender v knowing $C(s, v) = c$ *a priori* and choosing latency c throughout the execution.

Now the task is to find the average latency of BCSSELECT. Consider the number of rounds required before each update of $nextpacket_{\log c}$, upper bounding the average latency we are looking for. One BCS block has $L/2$ terms because the greatest BCS index $j = \log L$ is selected for the first time when $k = L/2$. According to Lemma 4.3.2 and 4.3.3, the total time cost to generate one block is exactly the time needed to get another $j = \log L$ after the previous $j = \log L$, given by:

$$\sum_{j=1}^{\log L - 1} \frac{L}{2} \cdot \frac{1}{2^j} \cdot 2^j + 2^{\log L} = \frac{L(\log L + 1)}{2}.$$

Lemma 4.3.2 tells us that latency c (BCS index $\log c$) appears every $2^{\log c} = c$ iterations of $schedule$. Therefore, $\log c$ appears $L/(2c)$ times in one block, and the average time needed for $nextpacket_{\log c}$ to update will be $O\left(\frac{L(\log L+1)/2}{L/(2c)}\right) = O(c \log L)$.

[1] Notice that the probability of selecting latency ℓ given by distribution π from RANDSELECT is equal to the proportion of latency ℓ in one such bounded BCS block.

In conclusion, the competitive ratio of BCSSELECT during one block is $O(\log L)$. Because every block is identical, this competitive ratio applies to the whole execution. □

4.3.3 LOWER BOUND

We will now prove that our algorithms, RANDSELECT and BCSSELECT, are close to optimal by showing that every blind rate adaption algorithm is at best $\Omega(\log L/\log\log L)$-competitive in single hop network with static links. Our argument will demonstrate that no sequence of rate selections can be sufficiently competitive for every receiver in a particular lower bound network. It applies to randomized solutions. It follows directly that the bound therefore holds for deterministic solutions as well.

Theorem 4.3.5. *For a randomized blind rate selection algorithm $\mathcal{A}$, if $\mathcal{A}$ is $f(L)$-competitive with respect to any single hop network and static links, then $f(L) \in \Omega\left(\log L/\log\log L\right)$.*

Assume for contradiction that we have some algorithm $\mathcal{A}$ that is $o\left(\log L/\log\log L\right)$-competitive for all single hop networks and static weight functions. For our lower bound, we define the following single hop *lower bound network*: let D be a directed graph that consists of $n = \log L/\log\log L$ receivers[2] denoted $r_1, r_2, ..., r_n$. Next, we define the weight function C such that for each $i \in [n]$, $C(s, r_i) = \log^i L$. That is, the weights in this network are: $W = \{\log L, \log^2 L, \log^3 L, ..., \log^{\log L/\log\log L} L = L\}$.

We proceed with a series of proof steps that build toward the conclusion that executing $\mathcal{A}$ in the lower bound network cannot yield the assumed small latency at every receiver. Our first step is to transform the algorithm $\mathcal{A}$ into an algorithm $\mathcal{B}$ that uses only the latencies in W. The following lemma states that there exist transformations of this type that do not affect performance.

[2]For notational simplicity we assume $\log L$ and $\log L/\log\log L$ are positive, integral values.

Lemma 4.3.6. *Let $\mathcal{A}$ be a rate selection algorithm that is $f(L)$-competitive in the lower bound network D. There exists an algorithm $\mathcal{B}$ that only selects latency in W but is still $f(L)$-competitive in the lower bound network.*

Proof. Given a latency ℓ, where $\log L < \ell < L$, we abuse the notation $\lfloor \ell \rfloor$ to indicate the largest value in W that is less than or equal to ℓ. For our transformation, we have algorithm $\mathcal{B}$ simulate $\mathcal{A}$ locally according to the following three cases:

1. If during a given round, $\mathcal{B}$'s local simulation of $\mathcal{A}$ has $\mathcal{A}$ broadcast a message m with latency $\ell \in W$, then $\mathcal{B}$ also broadcasts m at this rate.

2. If the simulation of $\mathcal{A}$ broadcasts a message m with a latency less than $\log L$, then $\mathcal{B}$ remains silent.

3. Finally, if the simulation broadcasts a message m with a latency ℓ between $\log L$ and L, such that $\ell \notin W$, then during the interval of ℓ rounds in which $\mathcal{A}$ is simulated as broadcasting m, $\mathcal{B}$ waits until the final $\lfloor \ell \rfloor$ rounds of this interval, and then broadcasts m with latency $\lfloor \ell \rfloor$ during this suffix.

Fix some execution of $\mathcal{B}$. We will now argue that every receiver receives the same packets in the same rounds in $\mathcal{B}$'s execution and in the simulation of $\mathcal{A}$. It will then follow that the competitive ratio for $\mathcal{B}$ is identical to $\mathcal{A}$.

We consider the above cases one by one. If Case 1 occurs, the receive behavior in $\mathcal{A}$'s simulation is preserved in $\mathcal{B}$'s execution, as $\mathcal{B}$ sends the packet at the same latency starting at the same round as in $\mathcal{A}$. If Case 2 occurs, then it is also clear that the receiver behavior stays the same as *no* receiver has a strong enough link to receive this message in the $\mathcal{A}$ simulation in the lower bound network, and in $\mathcal{B}$'s execution the packet is simply not broadcast. Finally, we consider Case 3. Let ℓ be the latency for the broadcast in the $\mathcal{A}$ simulation. By the definition of the lower bound network, only receivers with a latency

less than or equal to $\lfloor \ell \rfloor$ can receive this message (as there are no receivers with a latency greater than $\lfloor \ell \rfloor$, but less than or equal to ℓ). Therefore, when $\mathcal{B}$ broadcasts this message at $\lfloor \ell \rfloor$, the same receivers receive the message. And because $\mathcal{B}$ waits until the final $\lfloor \ell \rfloor$ rounds of the ℓ-round interval $\mathcal{A}$ uses in its simulation to begin its broadcast, these receivers get the message in the same round in both settings. □

Fix some blind rate selection algorithm $\mathcal{B}$ that only uses latencies in W. We now prove that a constant fraction of the packets received by a given receiver must be at a "good" rate (i.e., the link weight) for that receiver.

Lemma 4.3.7. *Let $\mathcal{B}$ be a rate selection algorithm that only selects latencies in W and is $f(L)$-competitive in the lower bound network D. Fix some duration $T_0 = L$. Let k_i be the number of messages received by receiver r_i in a T round execution of $\mathcal{B}$, where $T \geq T_0$. It follows that at least $\lceil k_i/2 \rceil$ of these messages are sent at latency $\ell_i = \log^i L$.*

Proof. Assume for contradiction that half or more of these k_i message were sent at a latency greater than ℓ_i. The minimum latency for a packet sent to receiver r_i is ℓ_i. Therefore, if we assume that at least half the messages arriving at r_i are sent at a latency longer than ℓ_i, the next best case average latency for r_i would be at least: $((k/2)\ell_{i+1} + (k/2)\ell_i)/k$.

In the above, we assume the minimum number of packets (in this case, $k/2$) were sent at a slower latency, and we made this the next slowest latency after ℓ_i (i.e., $\ell_{i+1} = \log^{i+1} L$). The rate above also assumes the source was only servicing r_i and sent packets continuously with no gaps throughout the T rounds. It is, in other words, a quite optimistic bound. We now simplify:

$$\frac{(k/2)\ell_{i+1} + (k/2)\ell_i}{k} = \frac{(k/2)\ell_i \log L + (k/2)\ell_i}{k} = \frac{(1/2) \cdot k \cdot \ell_i(\log L + 1)}{k} > \frac{1}{2} \cdot \ell_i \cdot \log L$$

The optimal average latency for r_i is clearly ℓ_i. Therefore, $\mathcal{B}$ is *at best* $(\log L/2)$-competitive. We assumed earlier, however, that $\mathcal{B}$ is $f(L)$-competitive for a function that is

no larger than $c \cdot \log L / \log \log L < \log L/2$ (for sufficiently small constant $c > 0$). This contradicts our assumption that at least half of r_i's packets were sent slowly. □

We have just established that to achieve a reasonable competitive ratio for a given receiver in our network, at least half of the packets sent to the receiver must use a rate well-suited to the receiver's link. We next establish formally another important observation for our overall lower bound: to achieve a good rate in an execution of length T, the source must successfully deliver many packets.

Lemma 4.3.8. *Let $\mathcal{B}$ be an algorithm that is $f(L)$-competitive in the lower bound network D. Fix some receiver r_i and duration $T \geq T_0 = L$. It follows that r_i receives at least $T/(\ell_i \cdot f(L))$ packets during these T slots, where $\ell_i = \log^i L$.*

Proof. Fix some $f(L)$-competitive algorithm $\mathcal{B}$, as well as some r_i and $T \geq T_0 = L$. Consider a T-round execution of $\mathcal{B}$ in the lower bound network. Let $\ell_1^*, \ell_2^*, ..., \ell_j^*$ be the latency for each receive event at r_i in our T-round interval. Let α be the average latency at r_i in this interval. Notice, by definition: $\alpha = (1/j) \cdot \sum_{h=1}^{j} \ell_h^*$. Also note, however, that by definition: $\sum_{h=1}^{j} \ell_h^* = T$. It follows that $\alpha = T/j$, and therefore $j = T/\alpha$. By assumption, $\mathcal{B}$ is $f(L)$-competitive. This means that when considering r_i in particular, its average latency is no greater than $\ell_i \cdot f(L)$ and $j \geq T/(\ell_i \cdot f(L))$, as required. □

We are now ready to prove our main theorem.

Proof (of Theorem 4.3.5). Let $\mathcal{A}$ be the rate selection algorithm that we assumed to be $f(L)$-competitive for some $f(L) < c \cdot \log L / \log \log L$. Let $\mathcal{B}$ be the constrained rate selection algorithm provided by Lemma 4.3.6. By the guarantees of this lemma, $\mathcal{B}$ is $f(L)$-competitive in the lower bound network D. We will now show that this leads to a contradiction.

In particular, we will show that for any sufficiently large duration $T \geq T_0 = L$, $\mathcal{B}$ is at best $\Omega(\log L)$ competitive, which is $\omega(f(L))$, which contradicts our assumption that it is $f(L)$-competitive.

To get this result, let k_i be the number of packets that the source sends at latency $\ell_i \in W$ in a T-round execution of $\mathcal{B}$ in the lower bound network D. By Lemma 4.3.8, we know that receiver r_i receives at least $T/(\ell_i \cdot f(L))$ packets. By Lemma 4.3.7, we know at least half these packets are sent at latency ℓ_i. It follows that $k_i \geq T/(2 \cdot \ell_i \cdot f(L))$. We can now evaluate how many rounds are required for the source to send the needed number of packets at each rate, and derive the following answer:

$$\sum_{i=1}^{n} k_i \cdot \ell_i = \sum_{i=1}^{\log L/\log\log L} \frac{T}{2 \cdot \ell_i f(L)} \cdot \ell_i = \frac{\log L}{\log\log L} \cdot \frac{T}{2f(L)}.$$

By assumption, however, $f(L) < c \cdot \log L/\log\log L$. If we set c to be sufficiently small (e.g., $c < 1/2$), it simplifies to something strictly larger than T. There are only T rounds available, however, to complete all broadcasts. This yields a contradiction to our assumption about the bound on $f(L)$, and therefore $f(L)$ is in $\Omega(\log L/\log\log L)$ □

4.4 Single Hop Network with Fading links

Now we turn our attention to the fading setting where link quality can change from round to round. We are going to show that no rate selection strategy can be competitive in the presence of fading links. In more detail, we will provide a lower bound of $\Omega(L)$ for deterministic rate selection algorithms and $\Omega(\sqrt{L})$ for randomized rate selection algorithms (this lower bound for randomized algorithms also holds even for non-blind rate selection in which the sender learns the fate of each packet).

4.4.1 Lower Bound for Deterministic Algorithms

A deterministic blind rate selection algorithm can be described as a fixed sequence of latency choices. Here we prove that for any such sequence we can define a fading link weight function for a two-node network (the simplest possible network for rate selection) that guarantees a poor competitive ratio. At a high-level, this argument defines a weight function that keeps the link weight large when the algorithm attempts fast transmissions, and reduces the weight to something small when the algorithm attempts slow transmissions.

Theorem 4.4.1. *For a deterministic blind rate selection algorithm $\mathcal{A}$, if $\mathcal{A}$ is $f(L)$-competitive with respect to two-node networks and fading links, then it follows that $f(L) \in \Omega(L)$.*

Proof. Fix a deterministic algorithm $\mathcal{A}$. For a given $n \geq 1$, let $\ell_{\mathcal{A}}(n)$ be the latency $\mathcal{A}$ uses for its n^{th} transmission. Fix any duration $T \geq L$. We will define a fading weight function $C_{\mathcal{A}}$ that will generate an average latency of at least $c \cdot L \cdot OPT$ during the first T rounds when $\mathcal{A}$ is run in a two-node network. (Notice, in the following use the notation $C_{\mathcal{A}}(r)$ to refer to the minimum latency on the single link in the network during round r. That is, for simplicity, we omit the link variable from our channel notation.)

Next, we define $X^*_{\mathcal{A}} = \{\ell^*_{\mathcal{A}}(r)\}_{r=1}^{T}$ to be the T round sequence where each $\ell^*_{\mathcal{A}}(r)$ describes the latency at which the sender is transmitting during round r. The definition of $\ell^*_{\mathcal{A}}$ follows from $\ell_{\mathcal{A}}$. That is, if the sender begins its n^{th} transmission at round r, then $\ell^*_{\mathcal{A}}(r+k) = \ell_{\mathcal{A}}(n)$ for $0 \leq k \leq \ell_{\mathcal{A}}(n) - 1$.

We consider two different cases with respect to $X^*_{\mathcal{A}}$ to define $C_{\mathcal{A}}$:

Case 1. No more than $T/2$ vales in $X^*_{\mathcal{A}}$ are 1s. In this case, consider the weight function $C_{\mathcal{A}}(r)$ defined by the adversary as:

$$C_{\mathcal{A}}(r) = \begin{cases} 1 & \ell^*_{\mathcal{A}}(r) > 1 \text{ and } 2 \mid r \\ L & \text{otherwise} \end{cases}.$$

Some observations about this weight function:

The transmission through algorithm $\mathcal{A}$ never succeeds unless transmission latency is L. Then the total number of packets conveyed by algorithm $\mathcal{A}$ is no greater than T/L, and the average latency is at least $T/(T/L) = L$.

Moreover, if latency ℓ is not 1, then $\lfloor \ell/2 \rfloor$ more packets will be sent by the optimal algorithm during these ℓ slots. Since there are more than $T/2$ non-one latencies in $X^*_{\mathcal{A}}$, the optimal algorithm is able to send more than $\lfloor T/4 \rfloor$. The optimal average latency OPT is therefore bounded by a certain constant $\delta > 0$.

Finally, the competitive ratio of algorithm $\mathcal{A}$ is therefore $\Omega\left(L/\delta\right) = \Omega(L)$.

Case 2. More than $T/2$ latencies in $X^*_{\mathcal{A}}$ are 1s. In this case, consider the adversary that chooses $C_{\mathcal{A}}(r)$ like this. The adversary scans $X^*_{\mathcal{A}}(r)$ from left to right until it encounters a 1 at some position r'. At this point, the adversary sets $C_{\mathcal{A}}(r) = L$ for all $r < r'$ if $C_{\mathcal{A}}(r)$ has not been defined yet. Then

$$(C_{\mathcal{A}}(r'), C_{\mathcal{A}}(r'+1)) = \begin{cases} (2,2) & \ell^*_{\mathcal{A}}(r'+1) = 1 \\ (L,1) & \text{otherwise} \end{cases}.$$

Then the adversary restarts the scan at position $r' + 2$. Once this is over, the adversary goes back and fills in all undefined values with L.

During the execution of algorithm $\mathcal{A}$, latency L is still the only latency that will succeed in transmission, since the only two non-L latencies the adversary adds are 1 and 2. When a 2 is added to the channel, this is when the algorithm is sending at latency 1, and when a 1 is added, it has an L on either side and the algorithm is sending at a slower latency than 1 during the transmission. This yields an average latency of more than T/L. Moreover, every latency $\ell^*_{\mathcal{A}}(r') = 1$ is paired. The optimal algorithm succeeds in sending one packet each time one of these pairs is discovered. We will find more than $T/4$ of these pairs since there are more than $T/2$ latency 1s in $X^*_{\mathcal{A}}$. Optimally, at least $T/4$ packets are transmitted, and $OPT \leq T/(T/4) = 4$.

The competitive ratio of $\mathcal{A}$ is therefore $\Omega\left(L/4\right) = \Omega(L)$. □

4.4.2 A Lower Bound for Randomized Algorithms

Here we show that randomization cannot guarantee much advantage over determinism given fading links. The following theorem holds even for non-blind rate selection algorithms in which the sender learns the fate of each packet.

Theorem 4.4.2. *For a randomized blind rate selection algorithm $\mathcal{A}$, if $\mathcal{A}$ is $f(L)$-competitive with respect to two-node networks and fading links, then it follows that $f(L) \in \Omega(\sqrt{L})$ for any duration $T \geq 2\sqrt{L}$. This bound holds even with packet delivery acknowledgements.*

Proof. Divide the T rounds into identical blocks of length $2\sqrt{L}$. Label each block with $1, 2, \ldots, T/(2\sqrt{L})$. We use $C_{\mathcal{A}}(r)$ to denote the channel latency for round r. Let $t_i = 2(i-1)\sqrt{L} + 1$, the round where block i begins.

For the i^{th} block ($i = 1, 2, \ldots, T/(2\sqrt{L})$), the adversary randomly selects an index b_i from $[1, \sqrt{L} - 1]$ with uniform randomness, and sets $C_{\mathcal{A}}(r)$ to $\sqrt{L}$ if $r \in [t_i + b_i, t_i + b_i + \sqrt{L} - 1]$ and L otherwise. In other words, the adversary chooses $\sqrt{L}$ consecutive positions from each block, starting at round b_i in the block, and sets their weights to $\sqrt{L}$. The weights on the other positions will be L, the largest latency.

We have the following two observations about the adversary we specified:

- Since we fix $L > 4$ and therefore $\sqrt{L} < L/2$, it is impossible to have a time interval of length L during which the weight of the link remains L.
- Any transmission with latency $\ell \in [1, \sqrt{L})$ will never succeed, because the weight function of the link does not support latency ℓ. Moreover, any transmission with latency $\ell \in (\sqrt{L}, L)$ will never succeed, because the number of consecutive rounds that allow latency ℓ is less than ℓ.

In other words, $\sqrt{L}$ and L are the only two succeeding latency. The optimal algorithm will choose to transmit with latency $\sqrt{L}$ during the selected rounds in each block and remain silent during the rest of the block. Then at least $T/(2\sqrt{L})$ packets will be transmitted in the duration of T slots and therefore $OPT \in O(\sqrt{L})$.

Now we fix an arbitrary randomized rate selection algorithm $\mathcal{A}$ that $\mathcal{A}$ only uses latency $\sqrt{L}$ and L (which is not necessarily blind). We argue that $\mathcal{A}$ has a competitive ratio of $\Omega\sqrt{L}$ by studying a restricted form of $\mathcal{A}$, which we call $\mathcal{A}^*$, that ignores all transmissions except those at latency $\sqrt{L}$. We will prove that the competitive ratio of $\mathcal{A}^*$ is in $\Omega(\sqrt{L})$. By studying algorithm $\mathcal{A}^*$, we are ignoring those packets that could have been sent by algorithm $\mathcal{A}$ with latency L here, but we will add them back later.

We fix $\mathcal{A}^*$ such that the sender either tries to send a packet at latency $\sqrt{L}$ somewhere in the first half of the block, or it does not. If the latter occurs, the transmission will not succeed since the choice of where the weight $\sqrt{L}$ begins in the block is made uniformly

and independently from $\mathcal{A}^*$'s behavior. If the former occurs, then the transmission succeeds only if it selects the exact position where weight $\sqrt{L}$ begins, with probability no greater than $1/(2\sqrt{L}-1)$. Note that this is valid even if the sender is provided the feedback on previous packets and its choice of latencies. Let X_i be the random indicator that evaluates to 1 if and only if algorithm $\mathcal{A}^*$ starts the transmission starts in the first half of block i (which might be nowhere because the algorithm might not transmit in the first half of the block). Then we have $E[X_i] \leq 1/(2\sqrt{L}-1)$. Therefore, if Y is the sum of these X_i's over the T rounds, we have

$$E[Y] = E[\sum_{i=1}^{T/(2\sqrt{L})} X_i] = \sum_{i=1}^{T/(2\sqrt{L})} E[X_i] \leq \frac{T}{2\sqrt{L}(2\sqrt{L}-1)}.$$

According to Markov's inequality [99] that

$$\Pr\{X \geq a\} \leq \frac{E[X]}{a}$$

for any nonnegative random variable X and $a > 0$, we have

$$\Pr\left\{Y \geq c \cdot \frac{T}{L}\right\} \leq \frac{1}{c}$$

for some constant $c > 2$.

Let $Q = T/Y$ be the average latency. Then

$$\Pr\left\{Q \leq \frac{L}{c}\right\} = \Pr\left\{Y \geq c \cdot \frac{T}{L}\right\} \leq \frac{1}{c},$$

and therefore

$$\Pr\left\{Q \geq \frac{L}{c}\right\} \geq 1 - \frac{1}{c}. \tag{4.1}$$

On the other hand, if we apply Markov's equation again on random variable Q, we have

$$\Pr\left\{Q \geq \frac{L}{c}\right\} \leq \frac{c}{L} \cdot E[Q]. \tag{4.2}$$

If we combine Inequality (4.1) and Inequality (4.2), we will get $E[Q] \geq (L/c)(1-1/c)$. Given that $OPT \in O(\sqrt{L})$, the competitive ratio of $\mathcal{A}^*$ is in $\Omega(\sqrt{L})$.

Finally, we add back the packets $\mathcal{A}^*$ ignored that $\mathcal{A}$ could have sent (i.e., packets sent with latencies other than $\sqrt{L}$), and upper bound how much that could improve the expected average latency of $\mathcal{A}$ compared to $\mathcal{A}^*$. Note that if algorithm $\mathcal{A}$ chooses latency L, the maximum number of packets it can send will be T/L, by choosing latency L throughout T rounds. However, since the expected value of Y is already in $O(T/L)$, this only changes the number by at most a constant factor. □

4.5 MULTIHOP RATE SELECTION AND PACKET ORDER PRESERVATION

We now consider multicast in multihop networks with static links. In more detail, consider a source s that might not have a direct link to a designated receiver t. In this case, s has to forward messages through intermediate nodes to get to t. We call this scenario a multihop network. We will describe and analyze an algorithm called MULTIBCSSELECT that works well in this setting with respect to the average latency possible on the best single path to each receiver. In more detail, we will prove that MULTIBCSSELECT achieves a competitive ratio of $O(\log L)$ with respect to this single path optimal solution.

The next question is to ask whether MULTIBCSSELECT is still competitive compared to multi-path optimality where packets will be transmitted along different s-t paths in parallel. Notice, once multiple paths are used, it might be possible to gain more performance, for example by routing different packets along different s-t paths. Our MULTIBCSSELECT cannot guarantee multi-path optimality. We will show, however, that in some sense, this is unavoidable, by proving that multi-path optimality cannot be achieved while also maintaining reasonable assumptions on packet ordering.

4.5.1 THE MULTIBCSSELECT ALGORITHM

Here we describe a blind rate selection algorithm for multihop transmission, called MULTIBCSSELECT. This algorithm is run by intermediate nodes (i.e., non-source nodes). The sours s, however, will execute the existing BCSSELECT logic (see Section 4.3.2). In more detail, MULTIBCSSELECT has intermediate nodes initialize their packet queues as empty queues, and a newly received packet will be pushed onto the back of each of the queues. Nodes will synchronize their sampling of the binary carry sequence (BCS) by attaching the current global index to the transmitted packets.

MULTIBCSSELECT (for intermediate nodes)

Initialization:
  for $j \leftarrow 1$ **to** $\log L$ **do**
    initialize Q_j with an empty packet queue

on receiving (p_0, k_0):
  $k \leftarrow$ UPDATE(k_0)
  for $j \leftarrow 1$ **to** $\log L$ **do**
    $Q_j \leftarrow push(Q_j, p_0)$
    $nextpacket_j \leftarrow peek(Q_j)$

Transmission:
  do
    $j \leftarrow schedule(k)$
    send $(nextpacket_j, k)$ with latency $\ell = 2^j$
    $pop(Q_j)$
    $nextpacket_j \leftarrow peek(Q_j)$
    $k \leftarrow$ UPDATE(k)
  while TRUE

Figure 4.3: The algorithm of MULTIBCSSELECT. See Figure 4.2 for the definition of UPDATE subroutine.

We will compare the average latency of MULTIBCSSELECT with *single path optimality*. In particular, we consider every path from s to t, and compare the performance of MULTIBCSSELECT to the best possible algorithm on this path, keeping the worst case performance over all paths as MULTIBCSSELECT's competitive ratio. We are going to show that MULTIBCSSELECT also achieves $O(\log L)$-competitiveness with respect to the single path optimal solution.

ALGORITHM

The blind rate selection algorithm for multihop packet transmission we will describe, MULTIBCSSELECT, is based on BCSSELECT. In particular, we have the source node s

run BCSSELECT, as in the single hop setting. The non-source nodes, by contrast, run the MULTIBCSSELECT algorithm described in Figure 4.3. This algorithm initializes each Q_j at an intermediate node as an empty queue. As an intermediate node receives a packet for the first time, it pushes it onto the back of each of its queues. This algorithm has nodes sample queues as in the single hop algorithm. In the case that it samples an empty queue, the node will simply transmit an "empty packet" (technically, we can interpret this as not sending any packet for the number of rounds required for the current latency). We synchronize the indexes nodes use to sample the BCS by propagating the current index in the transmitted packets. This index is initialized by the source so once every node has received a packet, all nodes are synchronized.

ANALYSIS

It is straightforward to show that MULTIBCSSELECT is correct in a multihop setting (consider only the queue associated with the slowest latency: every packet at every node is sent at the slowest latency from this queue in their original order that they arrive at the source). More interesting is analyzing its performance.

Because we consider single path optimality, we restrict our attention to a subgraph P consisting of any path from s to some fixed destination t, i.e., $V_P = \{v_0 = s, v_1, v_2, \ldots, v_n, v_{n+1} = t\}, E_P = \{(v_i, v_{i+1}) : i = 0, 1, \ldots, n\}$. We show that the performance of MULTIBCSSELECT is competitive with the optimal possible performance on every such P.

Theorem 4.5.1. *The* MULTIBCSSELECT *blind rate selection algorithm is* $O(\log L)$*-competitive with respect to the single path optimal solution.*

Before completing the proof of Theorem 4.5.1, we will bound the performance of the optimal algorithm on P:

Lemma 4.5.2. *Fix some $n \geq 0$ and a multihop route consisting of the path P with $n+2$ nodes $v_0, \ldots, v_{n+1}$, where $s = v_0$ and $t = v_{n+1}$, and also fix a static link weight function C. Suppose $c^* = \max_{0 \leq i \leq n}\{C(v_i, v_{i+1})\}$. The average optimal latency of packets sent from s to t on P is in $\Omega(c^*)$.*

Proof. Note that packets will be queued at bottlenecks where the links have poor quality. After passing the last bottleneck, packets will never be queued again, and will arrive at the destination at the slowest rate in the best case.

In more detail, let v_β be the last node on the path P such that $C(v_\beta, v_{\beta+1}) = c^*$. Packets cannot be delivered from v_β to $v_{\beta+1}$ at a rate faster than one packet every c^* rounds. Therefore, no node v_i with $i > \beta$ on the path can receive packets on this path at a rate faster than one every c^* rounds. It follows that in T rounds, t receives at most T/c^* packets, leading to an average latency of c^*. □

We now prove the main theorem.

Proof (of Theorem 4.5.1). We analyze the average latency of MULTIBCSSELECT. Consider some link $(v_\beta, v_{\beta+1})$ with β being the greatest index such that $C(v_\beta, v_{\beta+1}) = \max_{0 \leq \alpha \leq n}\{C(v_\alpha, v_{\alpha+1})\} = c^*$. In other words, $(v_\beta, v_{\beta+1})$ is the last bottleneck, or the last slowest link.

Consider only the queue associated with latency c^*. At the source, this queue holds all packets in order. By induction we will see that every time this queue comes up in the sampling, every node v_i ($i \leq n$) that has a non-empty queue for c^* delivers a new packet to v_{i+1}. As established in the proof of Theorem 4.3.4, this simply occurs once every $c^* \log L$ rounds. It follows that once every queue associated with this speed on the path is non-empty (which occurs after at most $c^* n \log L$ rounds), then the destination receives a new packet from this queue once every $c^* \log L$ rounds. Therefore, the corresponding competitive ratio on this path is in $O(\log L)$, given the optimal average latency of c^* (see Lemma 4.5.2).

Since this is true for all paths P, including the best such path, we have proved our claim. □

4.5.2 Impossibility of Multiple Path Optimality with Packet Order Preservation

We previously proved that MultiBCSSelect provides an average latency at each receiver that is close to optimal with respect to the single best path from the source to that receiver. It is possible, however, that an algorithm that sends different packets on different paths in parallel to a single receiver might perform better at that receiver. Here we prove it is not possible to be both close to optimal with respect to these multi-path solutions while also maintaining a natural order preserving property.

Generally speaking, a rate selection algorithm is order preserving if the order in which every packet is received is not too different from its order in the source's packet queue. It is not difficult to show that our MultiBCSSelect algorithm is perfectly order preserving. We formalize this property with the following two definitions:

Definition 4.5.3. *We define the sequence number of a packet p, denoted by $seq(p)$, to be the order of packet p in the source's packet queue at the beginning of the execution (where the packet at the head of the queue occupies position 1, and so on). Fix some non-source node t. Similarly, we define the transmission number of p with respect to t, denoted by $tn(t, p)$, to be the order in which t first received p, ignoring duplicate receives of packets (e.g., if t receives $p, p, p, p,'' , p'', p', p$ in an execution, then $tn(t, p') = 3$).*

Definition 4.5.4. *Fix some integer $\delta \geq 0$. A rate selection algorithm is δ-order preserving if for every packet p in the source queue, and every non-source node t, we have $|seq(p) - tn(t, p)| \leq \delta$.*

This notion of δ-order preserving is important for many applications in which received packets need to be reordered for processing. If I need, for example, k out of an original group of t packets to recover some coded information, and some of these packets can get arbitrarily out of order, I might have to wait an arbitrarily long time to complete the decoding.

We continue by noting that our multihop algorithm is perfectly order preserving:

Theorem 4.5.5. MULTIBCSSELECT *is* 0*-order preserving.*

Proof. The source node copies its source queue into $\log L$ transmission queues, each one associated with a different latency. Packets are removed and transmitted from each queue in FIFO order. A straightforward consequence is that for any two packets p and p', such that $seq(p) < seq(p')$, the source cannot send p' for the first time before it sends p (consider the queue from which the source samples p' for the first time: in order to reach p' in that queue, the source must have previously sampled and transmitted p).

It then follows that all neighbors of the source will receive messages for the first time in the same order as they appear in the source queue. They will subsequently add them to their transmission queues the same way. We can, therefore, apply the same argument as before to show this order is preserved to their neighbors, and so on, until we have considered every node in the network. It follows that for any non-source node t and any two packets p and p, if $seq(p) < seq(p')$, then $tn(t, p) < tn(t, p')$. □

With these definitions established, we can now state our main theorem, which claims that a blind algorithm cannot be both non-trivially competitive with respect to the multiple path optimal results, and be order preserving for some fixed δ.

Theorem 4.5.6. *There exists a constant $c' > 1$, such that for every integer $\delta \geq 0$ and competitive factor $c < L/c'$, there does not exist a blind rate selection algorithm that is c-competitive with respect to the multiple path optimal solution* and *δ-order preserving.*

To prove this lower bound, we will make use of a graph $D_r = (V, E)$, where V consists of a source, s, a destination, t, and L relay nodes, $r_1, r_2, \ldots, r_L$. Let $E = \{(s, r_i) : 1 \leq i \leq L\} \cup \{(r_i, t) : 1 \leq i \leq L\}$. Fix $C(s, r_i) = 1$ and $C(r_i, t) = L$ for all $i = 1, 2, \ldots, L$.

Fix some rate selection algorithm $\mathcal{A}$ that is c-competitive for some constant $c \geq 1$. To prove Theorem 4.5.6, we will show that there exists a network such that for all $x \geq 1$, there exists a packet p such that $|seq(p) - tn(t, p)| = \Omega(x \cdot L)$. For any fixed δ, therefore, we can find a sufficiently large x for which the algorithm is not δ-order preserving.

Let us study the constant competitive algorithm $\mathcal{A}$. Since all links coming out of the source have the same capacity, relay nodes will receive the same packet in the same transmission round. In order to achieve good competitiveness, relay node may not send packets in FIFO order. Otherwise, there will be a great amount of repetition of packets at the destination t, since relay nodes receive the same packet in each round. Actually we can claim without proof that the constant competitive solution for one single transmission round is to have L different relay nodes send $\Theta(L)$ different packets in the queue.

We will prove that after x transmission rounds (xL communication rounds), the maximum sequence number of the packets that have already been sent will be $\Omega(x) \cdot \Theta(L) = \Omega(xL)$ for all $x \geq 1$. We will forget about the first $L + 1$ communication rounds and regard the start of the $(L + 2)^{\text{th}}$ rounds as the start of $x = 1$.

Lemma 4.5.7. *When executing $\mathcal{A}$ on graph D_r, there exists some relay node r, such that for every integer $x \geq 1$, there exists an integer $x' \geq x$, such that $seq(p_{x'}^{(r)}) \geq (1/c)x'L$, where $p_{x'}^{(r)}$ is the x'^{th} unique packet that r sends.*

Proof. Assume for contradiction that there exists some $x \geq 1$ such that for every relay node r_i and every $x' \geq x$, $seq(p_{x'}^{(r_i)}) < (1/c)x'L$. Consider the first x unique packets that each relay node sends. Let w be the largest sequence number among these packets. Then

we have $x \leq w \leq cw$, because otherwise there exists two among the first x packets that having the same sequence number.

Fix $x' = \max\{cw, T_0\}$, where T_0 comes from the definition of c-competitiveness (recall, Definition 4.2.1 requires the competitive ratio of c only for executions of length at least T_0). Let r be the first relay node to send x' unique packets (breaking ties arbitrarily). Let ℓ be the round when this final packet is done. We are going to prove that by round ℓ, the number of unique packets sent to destination t is less than $(1/c)x'L$.

By our contradiction assumption, the sequence number of the x'^{th} packet is less than $(1/c)x'L$ ($> w$), and it is valid for all packets that each relay node sent after the x^{th} unique packet. We know that the sequence numbers of the first x packets sent by each node are also less than $(1/c)x'L$, because we fixed w as the maximum value during these rounds. It then follows that at round ℓ, the largest sequence number yet sent is less than $(1/c)x'L$, and therefore, less than $(1/c)x'L$ unique packets have been sent total up to this point.

We next note that $\ell \geq x'L$, as it takes any individual node at least L rounds per unique packet sent. During this interval, the optimal algorithm in D_r can deliver $x'L$ unique packets to the destination. We just established, however, that our algorithm has delivered less than a factor of c of this amount. This contradicts our assumption that our algorithm is always c-competitive. □

A simple counting argument yields the next lemma:

Lemma 4.5.8. *When executing $\mathcal{A}$ in any network, for every node u, after u transmits δ unique packets, the smallest sequence number among packets u has not yet sent is no more than $\delta + 1$.*

Proof. If packets are sent in order, then the smallest sequence number for unsent packets will be $\delta + 1$. Otherwise, at least one of packets numbered from 1 to δ is skipped, and the smallest sequence number for unsent packets will be no greater than δ. □

We can now pull together the pieces to prove our main theorem.

Proof (of Theorem 4.5.6). The key observation used by this proof is that a relay node r_i cannot distinguish an execution in D_r from an execution in the graph $D_r^{(i)}$ which consists only of: the source s, the only relay node r_i, the destination t, a directed edge (s, r_i), and a directed edge (r_i, t). Now consider an execution of $\mathcal{A}$ in L copies of $D_r^{(i)}$, one for each r_i. At the same time, run this algorithm with the same random bits in D_r. We will look at the behavior of $\mathcal{A}$ in D_r to point to an i for which $D_r^{(i)}$ behaves poorly.

In more detail, we apply Lemma 4.5.7, which identifies some r_i in D_r for which the lemma statement holds. Let x be the value identified by the statement for r_i. Let $x' \geq \max\{x, (\delta+1)/(\frac{L}{c}-1)\}$, where δ is the order-preserving bound from the theorem statement. Consider $p_{x'}^{(i)}$, the x'th packet sent by r_i. By the statement, $seq(p) \geq (1/c)x'L$. By Lemma 4.5.8, however, there is some sequence number $q \leq x'+1$, such that r_i has not yet sent the packet with that number.

Now consider r_i in $D_r^{(i)}$. It too will send a packet with sequence number at least $(1/c)x'L$ before it sends a packet with number $x'+1$. Because r_i must eventually send every packet in this graph (as it is the only relay node), when it does eventually get to the packet with sequence number $x'+1$, it will be out of order. In particular, the gap between this packet's number and the x'th packet's number is at least: $x'(L/c) - q \geq x'(L/c) - (x'+1) > \delta$. (Notice, it is here that we require that c is sufficiently small compared to L.) We have just identified, however, an execution of $\mathcal{A}$ in a graph that is non-order preserving. A contradiction. □

CHAPTER 5

FAIR CONTENTION RESOLUTION ALGORITHMS ON MULTIPLE CHANNELS WITH COLLISION DETECTION

The contention resolution problem assumes multiple processes are connected to one or more shared channels. It also assumes that time proceeds in synchronized rounds. In each round, each process can transmit on at most one channel. If multiple processes transmit on the same channel, these messages will be lost due to collision. The problem is solved in the first round a process transmits alone on a channel, resolving the contention.

In this chapter, we will study new contention resolution algorithms with respect to not only their time complexity but also their "fairness". We define the fairness of contention resolution algorithms in an intuitive way: every active process will be selected as the final leader (i.e., be the first process to transmit alone) with similar probability. We explore efficient and fair contention resolution algorithms under the assumptions of both multiple channels and collision detection. The results presented in this chapter first appeared in [29].

In more detail, we describe and analyze two new fair contention resolution algorithms for the setting of $\mathcal{C} > 1$ channels and a total of n possible processes that might be activated and connected to the channel. The first algorithm solves the problem in an optimal $O\left(\frac{\log n}{\log \mathcal{C}} + \log\log n\right)$ rounds in the restricted case where only two processes are activated—exactly matching the best known lower bound [83]. The second algorithm solves the problem in a near optimal $O\left(\frac{\log n}{\log \mathcal{C}} + (\log\log n)(\log\log\log n)\right)$ rounds for the general case where any number of processes are activated—falling just shy of the lower bound by a $\log\log\log n$

factor when $\mathcal{C}$ is large (i.e., the $\log \log n$ term dominates) and matching the lower bound otherwise.

These results generate both specific and general impacts. The specific impact is that they close a long-standing open question regarding a classical distributed algorithm problem. The general impact comes from the new techniques we developed, called ***coalescing cohorts***, to prove our results. The idea is that when we first enter this step, the active processes are not coordinated. As we proceed with our basic strategy of conducting searches to reduce active processes, however, we *coalesce* coordinated groups of processes that we call *cohorts*. In particular, our coalescing cohorts strategy provides a method to quickly build large collections of coordinated processes in distributed multiple channel models. In this chapter, we use the structure in these groups to simulate efficient strategies from the parallel algorithms literature. This strategy can be combined with a variety of well-known parallel algorithms to speed up computation in our distributed model. Even without parallel algorithm simulation, however, we note that the structure provided by these cohorts still provides a powerful algorithmic tool that can potentially be leveraged in developing efficient solutions for many problems.

In the last section of this chapter, the fairness of contention resolution algorithms will be formally defined. We will prove that both our contention resolution algorithms are fair. We believe that a fair contention resolution algorithm might help distribute resources or workload evenly in a distributed system that uses contention resolution as a key subroutine.

5.1 THE CONTENTION RESOLUTION PROBLEM

We parameterize our model with integers $\mathcal{C} > 0$ and $n \geq 2$. We assume a set V consisting of n *processes*, each executing a piece of a distributed algorithm, and a collection of $\mathcal{C}$ multiple-access channels (called *channels* for short in the following), labeled $1, 2, ..., \mathcal{C}$. Our algorithms do not require processes to have unique IDs (though the lower bounds in [83] hold even if they do).

At the beginning of an execution, some arbitrary subset $A \subseteq V$, called *active set*, of the n possible processes are designated as *active.* (When specified, we sometimes consider a *restricted* case in which $|A|$ must equal 2.) The active processes are the only processes that participate in the execution. Time proceeds in synchronous rounds. In each round, each process in A must make two decisions: (1) it must choose a single channel from 1 to $\mathcal{C}$ on which to participate; and (2) it must decide whether to *transmit* a message or *receive*. Each individual channel behaves like a standard MAC with (strong) collision detection. In more detail, fix some process $u \in A$ that chooses to participate on channel i in round r. If no process transmits on i in this round, u detects silence. If exactly one process transmits on i in this round, u receives this message. If two or more processes transmit on i, u receives a collision notification. Notice, as in [83], we assume the classical definition of collision detection (common in much of the original work on this problem), where both transmitters and receivers learn about message collisions on their channel in a given round.

Here we give the formal definition of contention resolution problem:

Definition 5.1.1 (Contention resolution)**.** *Suppose $A \subseteq V$ is an active set. Then we say that **contention resolution** is solved in the first round in which some active process in A transmits on channel* 1.

We assume all processes start during the same round. It is easy to transform a solution that works in this model to a solution that works in the harder model where processes can

start during different rounds, at the cost of a factor of 2 in time complexity. In the non-simultaneous wake-up case, we can have each process listen for two rounds on channel 1. If both rounds are silent, it starts running a modified version of the protocol where each process transmits in the odd rounds (on channel 1) and runs the protocol in the even. If the process instead hears a collision or message in the first two rounds, it just stop participating in the algorithm.

In this thesis, we consider results that hold with high probability in n (the maximum number of processes that might be activated), which we define to mean probability at least $1 - 1/n^c$ for some fixed constant $c \geq 1$. We define the notation $[i, j]$, for integers $i \leq j$, to denote the range $\{i, i + 1, ..., j\}$, and define $[i]$, for integer $i > 0$, to denote $[1, i]$.

TWOACTIVE (for process $i \in A$)

(Step #1: ID Reduction)
  do
    choose ID_i from $[1, \mathcal{C}]$ with uniform randomness
    transmit on channel ID_i
  until i is alone on channel ID_i

(Step #2: Symmetry Breaking)
  $L \leftarrow$SPLITCHECK$(0, \lg \mathcal{C}, ID_i)$
  $p_L \leftarrow$ the tree node in level L on P_i
  if p_L is the left child of its parent at level $L - 1$
  then transmit on channel 1

SPLITCHECK(l, r, id)
(return the level where P_i and P_j split)

if $l \geq r$ **then return** l
else
  $m = \lfloor (l + r)/2 \rfloor$
  $p_m \leftarrow$ the tree node in level m on path P_{id}
  transmit on the channel corresponding to this node
  if collision is detected **then**
    return SPLITCHECK$(m + 1, r, id)$
  else return SPLITCHECK(l, m, id)

Figure 5.1: The TWOACTIVE algorithm.

5.2 CONTENTION RESOLUTION FOR TWO PROCESSES

We begin by considering the restricted case where exactly two processes are activated, i.e., $|A| = 2$. Under this assumption, we describe and analyze an algorithm that solves contention resolution in $O\left(\frac{\log n}{\log \mathcal{C}} + \log \log n\right)$ rounds, with high probability. This algorithm matches the lower bound from [83], which holds for this two-node assumption. Moreover, we will show in Section 5.4 that this algorithm satisfies our new fairness definition (see Definition 5.4.3). This algorithm is also useful as it isolates and highlights some of the general ideas leveraged by the more involved general algorithm that follows.

In the following we assume that $\mathcal{C}$ is a power of 2 (the strategies are easily modified to handle other values). We also assume $\mathcal{C} \leq n$. Notice, no optimality is lost by this latter assumption.[1]

5.2.1 THE TWOACTIVE ALGORITHM

Here we describe the TWOACTIVE algorithm (see Figure 5.1). The algorithm executes in two steps. The first step implements an ID reduction strategy that renames the two active processes with unique IDs from the range $[\mathcal{C}]$. The two processes each choose a channel from the $\mathcal{C}$ available channels and then transmit and use their collision detectors to see if they are alone. (This step leverages the strong collision detection assumption that allows transmitters to detect collisions.) The processes repeat this random channel selection strategy until they detect they are on unique channels: at which point, each process then adopts the label of its selected channel as its new unique id. This step requires $O(\log n / \log \mathcal{C})$ rounds (with high probability).

The second step makes use of these new IDs to efficiently break symmetry among the two processes. To do so, the processes consider a canonical full binary tree $T_{\mathcal{C}}$, with $\mathcal{C}$ leaves labeled with the values in $[\mathcal{C}]$. Note that $T_{\mathcal{C}}$ has a height of $h = \lg \mathcal{C}$. Let i and j be the two active processes, and let ID_i and ID_j be their IDs selected in the previous step. Process i considers the unique simple path in $T_{\mathcal{C}}$ from the root to the leaf labeled ID_i, and j does the same with respect to ID_j. We call these paths P_i and P_j, respectively. Notice that these paths begin together at the root and split at some point by the time they reach their respective leaves. This second step conducts a binary search over the h levels of the tree to find the smallest level at which the two paths diverge. This search requires an assignment of a unique channel to each tree node at the level being checked at a given

[1] The lower bound for $\mathcal{C} > n$ is $\Omega(\log \log n)$, and our algorithm executes in $O(\log \log n)$ rounds when run on n channels.

step of the algorithm; that is, the multiple channel assumption is necessary for this step as well. Once we have found this level—which we call ℓ in this exposition—breaking symmetry is simple: the single process that is a left child of its parent in $\ell - 1$ (on its path from the root) *wins* and transmits on channel 1 to solve the problem. This step requires $O(\log h) = O(\log \log \mathcal{C}) = O(\log \log n)$ rounds.

5.2.2 Analysis

We are now ready to state our main correctness theorem:

Theorem 5.2.1. *In a network with 2 active processes (out of n possible processes) and $\mathcal{C}$ channels,* TwoActive *solves contention resolution in $O\left(\frac{\log n}{\log \mathcal{C}} + \log \log n\right)$ rounds, with high probability.*

The correctness of our theorem is a direct consequence of the following two lemmas.

Lemma 5.2.2. *With high probability: processes i and j conclude Step #1 with $ID_i \neq ID_j$, during the same round $r \in O\left(\frac{\log n}{\log \mathcal{C}}\right)$.*

Proof. Fix $t = \lg n / \lg \mathcal{C}$. The probability that processes i and j select the same channel for t consecutive rounds in the ID reduction phase is bounded as:

$$\prod_{r=1}^{t} \Pr\{ID_i = ID_j \text{ for round } r\} = \prod_{r=1}^{t} \frac{1}{\mathcal{C}} = (2^{-\lg \mathcal{C}})^t = 2^{-\lg n} = \frac{1}{n},$$

which is sufficiently small for our lemma statement. (Notice, we can easily adjust this result for probability $1/n^c$, for constant $c > 1$, by simply increasing t by a factor of c.) □

Lemma 5.2.3. *Assume that processes i and j begin Step #2 during the same round with $ID_i \neq ID_j$, then this step will go on to solve contention resolution in an additional $O(\log \log n)$ rounds.*

Proof. Consider a binary array $B \in \{0,1\}^{\lg \mathcal{C}+1}$, defined such that for each $m \in [0, \lg \mathcal{C}]$, $B[m] = 1$ iff P_i and P_j pass through the same tree node in level m of $T_{\mathcal{C}}$. Notice that $B[0] = 1$, $B[\lg \mathcal{C}] = 0$, and the array, when read from small to large indices starts with 1's then changes over to 0's. The SPLITCHECK subroutine implements a standard binary search logic on B to identify $\ell = \min\{m : B[m] = 0\}$. The correctness of this search depends on the correctness of the logic SPLITCHECK uses to test whether a given position is 1 or 0. This correctness follows from how the algorithm assigns i and j to channels when testing a given level m. The processes then use collision detection to determine whether or not they share the same channel for a given check.

Once we have established that the processes effectively set $L = \ell$ with their call to SPLITCHECK, the correctness of the whole routine follows. By definition, i and j share a parent in level $\ell - 1$ in $T_{\mathcal{C}}$ and therefore only one of these two processes is the common parent's left child—the process that will go on to transmit alone on channel 1 and solve the problem.

Finally, we note that the time required to complete this search is in $O(\log h) = O(\log \log \mathcal{C}) = O(\log \log n)$, where the final step follows from our assumption that $\mathcal{C} \leq n$. □

5.3 CONTENTION RESOLUTION FOR ANY NUMBER OF PROCESSES

We now present our main algorithm SPLITELECT that solves contention resolution for any number of active processes. The SPLITELECT algorithm consists of three steps that are executed one after another in a synchronized manner. The first step leverages a straightforward *knock out* strategy to reduce the number of active processes to a count in $O(\log n)$. (This preliminary reduction will simplify the steps that follow.) The second step reassigns active processes unique IDs from the space $[\mathcal{C}/2]$ (further reducing the number of active processes if needed to enable this task). The third step leverages the guarantees of the two that precede it to solve the contention resolution problem. Our main theorem follows directly from Theorems 5.3.2, 5.3.3, and 5.3.14:

Theorem 5.3.1. *In a network with up to $n \geq 1$ possible active processes and $\mathcal{C} \geq 1$ channels,* SPLITELECT *solves contention resolution in $O\left(\frac{\log n}{\log \mathcal{C}} + (\log\log n)(\log\log\log n)\right)$ rounds, with high probability.*

5.3.1 STEP #1: REDUCE TO $O(\log n)$ ACTIVE NODES

The first step of SPLITELECT algorithm, which we call algorithm REDUCE (see Figure 5.2), reduces the number of active nodes to $O(\log n)$. In more detail, each iteration of REDUCE consists of two rounds. During each of the rounds, each active process broadcasts on channel 1 with probability $1/\hat{n}$ (initially $\hat{n} = n$), and otherwise receives on channel 1. If process v broadcasts alone, then v becomes the leader and terminates with contention resolved. If process v chooses to receive while some other process broadcasts on channel 1 (collided or not), it becomes inactive. Otherwise, if v's message collides with other messages, it remains active. After each iteration, processes that remain active increment the broadcast probability by square-rooting $\hat{n}$; i.e., $\hat{n} \leftarrow \sqrt{\hat{n}}$.

This reduction will prove helpful to both steps that follow. Note that with collision detection, reducing a set of no more than n processes to less than $O(\log n)$ processes does not require multiple channels. Accordingly, the algorithm we deploy for this purpose only uses channel 1. Our algorithm REDUCE uses a standard knock out strategy and reduces the active process count to $O(\log n)$ in $O(\log \log n)$ rounds. We formalize this result with the following theorem which can be shown using standard techniques from the single channel setting:

Theorem 5.3.2. *There exists a constant $\alpha \geq 1$, such that for any constant $\beta \geq 1$, when algorithm* REDUCE *terminates the number of active processes is between 1 and $\alpha\beta \log n$, with probability at least $1 - n^{-\beta}$.*

Proof. We first note that by the definition of the algorithm it is impossible for *all* participants to become inactive (as becoming inactive in a given round requires that there is a process that transmits in that round and therefore ends the round active). This satisfies the lower bound on active processes claimed by the theorem statement. The remainder of the proof therefore focuses on the upper bound.

We next define some useful notation. Fix some $t \in [T]$, where $T = \lceil \lg \lg n \rceil$ is the total number of iterations of the **do** loop. Consider the two-round iteration of the loop that begins with $r = t$. We define n_t to be the number of active processes at the start of this iteration, n'_t to be the number of active processes at the beginning of the second round of this iteration, and $\hat{n}_t$ to be the value of $\hat{n}$ at the beginning of this iteration. Our strategy in this proof is to prove that if n_t is sufficiently small for iteration t, then with high probability n_{t+1} will be sufficiently small for $t + 1$. We can then inductively apply this insight, using a union bound in each step to maintain high probability, until we get to the conclusion that n_T is sufficiently small—which will directly imply the needed upper bound on active processes. In more detail, consider this claim:

Assume that $n_t \leq c\hat{n}_t^2 \log n$, for a constant $c \geq 1$ that we will fix later. It follows that $n_{t+1} \leq c\hat{n}_{t+1}^2 \log n$, with high probability in n (where the exponent on n increases linearly with c).

To prove this claim, we first note that if $n_t \leq c\hat{n}_{t+1}^2 \log n = c\hat{n}_t \log n$ we are done. Moving forwarding, assume n_t is greater than this value. Let X_t be the number of processes that transmit in the first round of this iteration. We know $\mu = E[X_t] = n_t/\hat{n}_t$. Given our upper and lower bounds on n_t, it follows that $c \log n < \mu \leq c\hat{n}_t \log n$. Because the transmission decisions are independent between processes, we can express X_t as the sum of independent indicator variables describing active processes' transmission decisions, and therefore apply a Chernoff bound to achieve concentration. In particular, we use the following special form of the Chernoff bound: $\Pr(|X_t - \mu| \geq \mu/2) < 2e^{-\mu/12}$.

We call a round *good* if the number of transmitters is within $[\mu/2, (3\mu)/2]$. By our above Chernoff bound, the probability that the round we are considering is *not good* is less than $2e^{-\mu/12} \leq 2e^{(c/12)\log n}$. Notice, for any constant $c' \geq 1$, this probability is (loosely) upper bound by $1/n^{c'}$, for $c \geq 24c'$.

Consider what happens if this round is good. Assuming $c \geq 2$, it follows that there is at least one transmitter. Either we elected a leader (and are done), or there are only X_t processes left active. Furthermore, we know this number of active processes is no more than $(3/2)\mu \leq (3/2)c\hat{n}_t \log n$.

Assuming that we have a good round (and did not directly elect a leader), there are two cases to consider for the remainder of this iteration. The first case is that after the first good round we have $n'_t \leq c\hat{n}_t \log n$. In this case, our claim now holds and we are done. The second case is that after the first good round, we have $c\hat{n}_t \log n < n'_t \leq (3/2)c\hat{n}_t \log n$. In this second case, we turn our attention to the second round of iteration t to complete the necessary reduction. In particular, we can now update our bounds for μ for this second

round as follows: $c \log n < \mu \leq (3/2)c \log n$. Applying the same analysis as in the first round it holds: with high probability (that grows with c), we had at least one transmitter, but no more than $(9/4)c \log n$. Notice that $(9/4)c \log n < c\hat{n}_t \log n$ for all values t used in the algorithm. Therefore, if the second round is also good, we satisfy the claim.

Pulling together the pieces, we have shown the probability that our claim holds is at least as large as the probability that we have two good rounds in a row. By a union bound, the probability that at least one of the two rounds is instead bad is polynomially small in n with an exponent that increases linearly with c.

To complete our proof, we begin with the base case of $t = 1$. The precondition for our claim trivially holds for $t = 1$ as for this case $c\hat{n}_t^2 \log n > n$. We can now apply our above argument to show that the claim holds with high probability. Conditioned on the claim holding for $t = 1$, we can apply our argument again to show that with high probability it holds for $t = 2$ as well. Conditioned on this event, we can do the same for $t = 3$, then $t = 4$, and so on, until the maximum value of $t = T$. A union bound over these rounds provides that with high probability, after the final iteration (assuming we did not previously elect a leader), the number of active processes is no more than $c\hat{n}_T \log n = O(\log n)$. As established above, the exponent on n in the high probability bounds grows linearly with c, allowing us to decompose this upper bound as specified in the theorem statement for any desired exponent β. □

5.3.2 Step #2: Reduce the Unique ID Space to $[\mathcal{C}/2]$

Our goal in this second step is to continue to reduce the set of active processes as needed until we succeed in reassigning processes unique IDs from $[\mathcal{C}/2]$. We accomplish this task

REDUCE (for active process v)

$\hat{n} \leftarrow n, r \leftarrow 1$
do
  repeat twice:
    v broadcasts on channel 1 with probability $1/\hat{n}$
    if v broadcasts without collision **then**
      v become leader and terminates
    else if v receives and does not hear silence **then**
      v becomes inactive and terminates
  $r \leftarrow r + 1$
  $\hat{n} \leftarrow \sqrt{\hat{n}}$
while $r \leq \lceil \lg \lg n \rceil$

Figure 5.2: The REDUCE algorithm.

with an algorithm we call IDREDUCTION, which guarantees to complete the needed reduction/renaming in $O(\log n / \log \mathcal{C})$ rounds, with high probability. We describe and analyze the algorithm below.

This algorithm repeatedly attempts to rename the active processes by having each such process select a channel from $\mathcal{C}/2$ with uniform randomness and then transmit to see if it is alone. If any processes are alone then they can adopt their channel label as their new unique ID, and then inform the rest of the processes that renaming is complete. The processes that did not yet adopt a new ID become inactive. If $\mathcal{C}$ is small compared to $\log n$ when we call IDREDUCTION, there may be too many processes for this renaming to succeed. To handle this case, the algorithm interleaves an aggressive reduction step similar to REDUCE, except we now use a fixed transmission probability $1/k$, for $k = \sqrt{\mathcal{C}}/144$. Our analysis, therefore, first focuses on showing that the reduction rounds efficiently reduce the number of active processes to a sufficiently small number compared to $\mathcal{C}$. It then focuses on showing that

the renaming will succeed quickly once we reach this level. We describe and analyze the algorithm below.

THE IDREDUCTION ALGORITHM

This algorithm alternates between *renaming* and *reduction* phases. The renaming phase is implemented by a pair of rounds. During the first round of the pair, all processes that are still active at the beginning of the renaming phase choose a channel from the range $[\mathcal{C}/2]$ with uniform randomness and transmit. If a process detects it is alone on some channel i, it adopts i as its unique id. In the next round of the pair all processes go to channel 1. Any process that adopted a unique id in the preceding round transmits. If there are any transmissions, the algorithm is over, and only those processes who transmitted remain active (with their recently adopted ids as their new unique id).

The reduction phase requires only a single round. During a reduction phase round, all processes that are still active transmit with probability $1/k$ on channel 1, where $k = \sqrt{\mathcal{C}}/144$. Processes that do not transmit receive on channel 1. If there is at least one transmission, then any active process that did not transmit becomes inactive. Otherwise, the set of active processes does not change in this round.

ANALYSIS

In the following, we use the notation A_r to represent the set of active processes in the beginning of the r^{th} round of IDREDUCTION. Note that $\ldots \subseteq A_{r+1} \subseteq A_r \subseteq \ldots \subseteq A_1 = A$. In the following, we assume $\mathcal{C}$ is a sufficiently large such that $k = \sqrt{\mathcal{C}}/144 \geq 3$. (Spread throughout this analysis are several other places where we similarly assume $\mathcal{C}$ is larger than some fixed constant.) We note that we are always safe to fix a constant lower bound for $\mathcal{C}$, as when $\mathcal{C} = O(1)$, the lower bound simplifies to $\Omega(\log n)$, which we can match with the well-known $O(\log n)$ contention resolution algorithm that is optimal for

the single channel case. Finally, to simplify notation, the theorem, lemma, and corollary statements that follow, when we claim a result holds *with high probability*, we mean that it holds with probability at least $1 - n^{-\beta}$, where $\beta \geq 1$ is a constant that increases along with the constants hidden within the asymptotic time complexity.

We begin our analysis by stating the main theorem. Its correctness follows directly from Corollary 5.3.5 and Lemma 5.3.7 which we describe and prove below.

Theorem 5.3.3. *Assume* $|A| = O(\log n)$. *With high probability:* IDREDUCTION *terminates in* $O(\log n / \log \mathcal{C})$ *rounds with no more than* $\mathcal{C}/2$ *active processes, each with a unique ID from* $[\mathcal{C}/2]$.

Our analysis proceeds in two pieces. In more detail, first we show that the reduction rounds reduce the number of active processes below $\mathcal{C}/6$ within $O(\log n / \log \mathcal{C})$ rounds. Second, we show that once we have less than $\mathcal{C}/6$ participants, the renaming rounds will succeed within an additional $O(\log n / \log \mathcal{C})$ rounds. The below lemma addresses the first piece:

Lemma 5.3.4. *Assume* $|A| = O(\log n)$. *There exists a round* $\hat{r} = O(\log n / \log k)$, *such that with high probability:* $|A_{\hat{r}}| \leq 24k \log k$.

Proof. Let $c_1 = 24$ be the constant from the lemma statement; i.e., our goal can be stated as achieving $|A_{\hat{r}}| \leq c_1 k \log k$. By assumption, $|A| \leq c' \log n$ for some constant $c' \geq 0$. Therefore, it clearly follows that $|A| \leq \gamma \log n$ for $\gamma = \max\{24, c'\}$. We also assume $k < n$. If this was not true, then the assumption that $|A| = O(\log n)$ would imply that $|A| \leq c' \log k$ which is smaller than $24k \log k$ for any constant c' once $\mathcal{C}$ is a sufficiently large constant, which would make the lemma trivially true. We also assume $\mathcal{C}$ is a sufficiently large constant that $k \geq 3$. An immediate corollary from this assumption is that $|A| \leq \gamma k \log n$. Because the set of active processes can never increase as the execution continues, this upper bound always holds.

To begin our proof argument, we note that in each round r for which we have not yet sufficiently reduced the number of active processes, we know: $c_1 k \log k < |A_r| \leq \gamma k \log n$. For each such round r, therefore, we can express $|A_r|$ as the quantity $\gamma k(\log n/q(r))$, for some value $q(r)$ that is from the range 1 to $T = (\gamma \log n)/(c_1 \log k)$. (We know $T > 1$ because $\gamma \geq c_1$ and $n > k$.) Furthermore, we know the sequence $q(1), q(2), q(3), ...$, is strictly non-decreasing, as the active process count can never grow.

Fix some reduction round r such that we are not yet done. Let X_r be the number of processes that choose to transmit during this round. We know: $E(X_r) = |A_r|/k = \gamma(\log n/q(r))$.

Because the transmission decisions are independent, we can express X_r as the sum of independent indicator variables describing the processes' transmission decisions, and can therefore apply a Chernoff bound to achieve concentration. We use the following special form:

$$\Pr[|X_r - E(X_r)| \geq E(X_r)/2] < 2e^{-E(X_r)/12}.$$

Notice, if $|X_r - E(X_r)| < E(X_r)/2$, then because $k \geq 3$ and $E(X_r) \geq 2$, it holds that we have reduced the set of active processes by at least a factor of 2; i.e., $|A_r|/|A_{r+1}| \geq 2$. (The former assumption ensures $(3/2)E(X_r)$ is not too large, and the latter ensures that $(1/2)E(X_r) \geq 1$.)

We want to bound the probability that $|A_r|/|A_{r+x}| < 2$, for some $x \geq 1$. That is, we want to bound the probability that after x rounds we still have not yet reduced the active process count at the start of the interval by at least a factor of 2. To bound this probability, we instead consider the *harder* model where if a round is not good, then no process becomes inactive. In this harder model, the probability that we have x reduction rounds in a row that are *not good* is less than:

$$\left(2\exp\left\{-\frac{E(X_r)}{12}\right\}\right)^x < \left(\exp\left\{-\frac{E(X_r)}{24}\right\}\right)^x = \exp\left\{-\frac{x\gamma \log e \ln n}{24q(r)}\right\},$$

where the first inequality loosely holds so long as we assume that $E(X_r) \geq 24$ (for all values of $q(r)$ from 1 to T the expectation is greater than this value). Next notice for that for $x \geq \frac{q(r)c_2 24}{\gamma \log e}$, defined for any constant $c_2 \geq 1$, this probability is less than n^{-c_2}. Put another way, starting from any given round r, with high probability, after $\Theta(q(r))$ reduction rounds we have reduced the active process count by a factor of at least 2. (And as we increase the constant c_2 hidden in the $\Theta(q(r))$ term, the exponent on the failure probability increases as well.)

We next partition reduction rounds into intervals, where the first interval starts in the first reduction round, and the start of interval $i+1$, for $i \geq 1$, corresponds to the first round after which the number of active processes is at least a factor of 2 smaller than at the start of interval i. Let r_i be the reduction round at the start of interval i. Let $\hat{i}$ be the latest interval before we fall below our target threshold of $c_1 k \log k$.

Consider the sequence of $q(r)$ values associated with the size of the active process sets during these intervals' start rounds: $q(r_1), q(r_2), ..., q(r_{\hat{i}})$. We note two things about this sequence. First, by our definition of an interval, for each i, $q(r_i) \leq q(r_{i+1})/2$. That is, the values at least double between intervals, corresponding to the process set size reducing by at least a factor of 2. We also note that by definition: $q(r_{\hat{i}}) = O(T) = O(\log n / \log k)$.

Leveraging our above probabilistic analysis, we note that for each interval i, with high probability, it takes only $O(q(r_i))$ rounds to get to interval $i+1$. Therefore, by a union bound, this is true of all intervals, also with high probability. Under this assumption, we can upper bound the number of rounds required to get through all $O(T)$ intervals with the summation $c + 2c + 4c + ... + T$, for some constant $c \geq 1$. This time complexity simplifies to $O(T) = O(\log n / \log k)$—as needed. □

Our goal is to get the number of active processes below $\mathcal{C}/6$. Notice, however, that for our definition $k = \sqrt{\mathcal{C}}/144$, then no more than $24k \log k$ processes guaranteed by Lemma 5.3.4 reduces to no more than:

$$\begin{aligned} 24k \log k &= 24(\sqrt{\mathcal{C}}/144) \log (\sqrt{\mathcal{C}}/144) < 24(\sqrt{\mathcal{C}}/144) \log (\sqrt{\mathcal{C}}) \\ &\leq (1/6)\sqrt{\mathcal{C}} \log (\sqrt{\mathcal{C}}) < \mathcal{C}/6. \end{aligned}$$

The following corollary is a direct implication of Lemma 5.3.4 and the above inequality and the fact $O\left(\frac{\log n}{\log k}\right) = O\left(\frac{\log n}{\log \mathcal{C}}\right)$:

Corollary 5.3.5. *Assume $|A| = O(\log n)$. There exists a round $\hat{r} = O\left(\frac{\log n}{\log \mathcal{C}}\right)$, such that with high probability: $|A_{\hat{r}}| < \mathcal{C}/6$.* □

We are left now to show that if the number of active processes reduces below $\mathcal{C}/6$, renaming becomes likely to succeed. Before making our main argument to this effect, we first isolate a useful balls-in-bins claim that the argument will leverage.

Lemma 5.3.6. *Throw b balls into m bins such that $b = m/\beta$, where $3 \leq \beta < m$. The probability that **no** ball ends up alone in a bin is less than $\frac{1}{2^{b/2}}$.*

Proof. To achieve our bound we will bound the probability that $(m - m/(2\beta))$ of m bins are empty. Notice, this event must hold if no ball ends up alone, as if more than $m/(2\beta)$ bins are full, then by a straightforward counting argument, at least one ball must end up alone.

Continuing with our calculation, consider a fixed set of $(m - m/(2\beta))$ empty bins. For each ball, the probability that the ball misses these empty bins and falls into one of the $m/(2\beta)$ *free* bins it is allowed to occupy, is $1/(2\beta)$. The probability that *all* balls land in a free bin is therefore $1/(2\beta)^{m/\beta}$. This probability concerns only a single set of empty bins. We now consider all such sets. In more detail, there are $\binom{m}{m-m/(2\beta)}$ different ways to select

our $(m - m/(2\beta))$ empty bins. Taking a union bound over these different selections, we derive that the probability that there is at least one selection of $(m - m/(2\beta))$ empty bins that remains empty after throwing m/β balls, is upper bounded as follows:

$$\frac{\binom{m}{m-\frac{m}{2\beta}}}{(2\beta)^{\frac{m}{\beta}}} = \frac{\binom{m}{\frac{m}{2\beta}}}{(2\beta)^{\frac{m}{\beta}}} \leq \frac{(2\beta e)^{\frac{m}{2\beta}}}{(2\beta)^{\frac{m}{\beta}}} = \left(\frac{e}{2\beta}\right)^{\frac{m}{2\beta}} < \frac{1}{2^{\frac{m}{2\beta}}},$$

where the last step holds because we assume $\beta \geq 3 > e \Rightarrow e/(2\beta) < (1/2)$. To achieve the bound stated in the lemma our final step is to substitute the definition $b = m/\beta$. □

We now leverage Lemma 5.3.6 to argue that renaming will succeed once the number of active processes is small enough for the above balls-in-bins argument to give us a sufficiently high probability of success. The probability we seek is something at least $(1-(1/\mathcal{C}))$. If we succeed with this probability, then the probability we fail $O(\log n/\log \mathcal{C})$ times in a row becomes small in n,

Lemma 5.3.7. *Fix a round r such that $|A_r| \leq \mathcal{C}/6$. With high probability, renaming succeeds and the algorithm terminates within $O(\log n/\log \mathcal{C})$ rounds.*

Proof. In renaming rounds, active processes each select a channel with uniform randomness. If any process is alone on its chosen channel the algorithm terminates. We will prove here that once we fall below $\mathcal{C}/6$ processes, this will occur with high probability after the stated number of attempts. To do so, fix some renaming round t. Let $n' = |A_t|$ be the number of processes still active during this round. Assume $1 < n' \leq \mathcal{C}/6$ (the lower bound is safe to assume as if $n' = 1$ it is trivial to show we will terminate in this round). Let $\hat{\mathcal{C}} = \mathcal{C}/2$. We consider two cases concerning the size of n', and will show for each that the probability we do *not* terminate in t is less than $1/2^{\log \hat{\mathcal{C}}/2}$. We will then show this probability is sufficiently low to terminate within the time claimed by the lemma statement with high probability.

The first case for the size of n' is when $n' \leq \sqrt{\hat{\mathcal{C}}}$. Fix any process $i \in A_t$. Let x be the channel it selects in this round. The probability that some other $j \in A_t$ also chooses x is $1/\hat{\mathcal{C}}$. By a union bound, the probability that at least one process in A_t chooses x is less than: $n'/\hat{\mathcal{C}} \leq 1/\sqrt{\hat{\mathcal{C}}} = 1/2^{\lg(\sqrt{\hat{\mathcal{C}}})} = 1/2^{\lg \hat{\mathcal{C}}/2}$, as needed.

The second case is when $n' > \sqrt{\hat{\mathcal{C}}}$. Here we can apply Lemma 5.3.6 to the problem of throwing n' balls in $\hat{\mathcal{C}}$ bins (which requires our assumption that $1 < n' \leq \mathcal{C}/6$ which implies $n' \leq \hat{\mathcal{C}}/3$). This lemma tells us that the probability that *no* ball is alone is less than $\frac{1}{2^{n'/2}}$. Given our assumption that $n' > \sqrt{\hat{\mathcal{C}}}$, it follows that $1/2^{n'/2} < 1/2^{\sqrt{\hat{\mathcal{C}}}/2} < 1/2^{\log \hat{\mathcal{C}}/2}$. This final step requires the assumption that $\log \hat{\mathcal{C}} \leq \sqrt{\hat{\mathcal{C}}}$, which is always true for $\hat{\mathcal{C}} \geq 16$. Given that we assumed $\mathcal{C} \geq 32$ at the beginning of this section, it follows that $\hat{\mathcal{C}} \geq 16$.

We have just shown that in every round (once the number of active processes is sufficiently small), we fail to terminate with probability less than $1/2^{\log \hat{\mathcal{C}}/2}$. The probability that we fail to terminate for $T = (c \log n)/\log \hat{\mathcal{C}}$ renaming rounds in a row (for a fixed constant $c \geq 1$), therefore, is less than:

$$\left(\left(\frac{1}{2}\right)^{\frac{\log \hat{\mathcal{C}}}{2}}\right)^{T} = \left(\left(\frac{1}{2}\right)^{\frac{\log \hat{\mathcal{C}}}{2}}\right)^{\frac{c \log n}{\log \hat{\mathcal{C}}}} = \left(\frac{1}{2}\right)^{\frac{c \log n}{2}} = \left(\frac{1}{2}\right)^{\frac{c}{2}}.$$

Because $T = O(\log n/\log \mathcal{C})$, it follows that the probability we fail to terminate in $T = O(\log n/\log \mathcal{C})$ rounds is polynomially small in n (with an exponent that increases with the constant c in T). □

Proof of Theorem 5.3.3. By assumption, when we start IDREDUCTION: $1 \leq |A| \leq O(\log n)$. We can, therefore, apply Corollary 5.3.5, which provides that with high probability the active process count reduces to less than $\mathcal{C}/6$ in $O(\mathcal{C} \log \mathcal{C})$ rounds. We can then apply Lemma 5.3.7, which provides that with high probability, we terminate in no more than $O(\frac{\log n}{\log \mathcal{C}})$ additional rounds. Adding together these three time complexities provides a total time complexity in $O(\frac{\log n}{\log \mathcal{C}})$, as needed.

As specified at the beginning of this section, for all three of these high probability bounds, we can increase the constant exponent on n by increasing the constant in the corresponding lemma's stated time complexity. Therefore, for any desired exponent for Theorem 5.3.3, we can increase the probability on our three lemmas such that after applying a union bound, the probability any fail to provide their stated guarantee is still sufficiently low. □

5.3.3 STEP #3: ELECT A LEADER

The result of the first and second step of SPLITELECT is that we now have $x \leq \mathcal{C}/2$ active processes with unique IDs in $[\mathcal{C}/2]$, and in addition $x = O(\log n)$ with high probability. These processes participate in the third and final step of SPLITELECT, which we call LEAFELECTION. This algorithm *deterministically* elects a leader using a tree of channels (i.e., a tree with $\leq \mathcal{C}$ nodes for which we have assigned a unique channel for each tree node). This algorithm runs in $O(\log h \log\log x)$ rounds, where x is the starting number of active processes and $h = \lg \mathcal{C}$ is the height of the tree of channels. As before, we shall assume without loss of generality that $\mathcal{C}$ is a power of 2.

ALGORITHM DESCRIPTION

Before describing the algorithm mechanics (see Figure 5.3 for pseudocode), we first discuss the tree of channels (or channel tree equivalently) the algorithm uses. As in Section 5.2 (the two process algorithm), we map channels to a complete binary tree T with $\mathcal{C}/2$ leaves, with channels identified in the same canonical fashion as before. In certain rounds, it will also be convenient to choose a single channel to represent each level (or row) in the tree; to do so, we choose the leftmost tree node at that level to act as the level's representative channel.

The core behavior of LEAFELECTION is to *coalesce* active processes into larger and larger groups we call *cohorts* such that every process in the same cohort has a distinct ID

LEAFELECTION (for active process v)

$cSize \leftarrow 1$ // size of all active cohorts
$cNode \leftarrow v$'s leaf // the subtree for v's cohort
$cID \leftarrow 1$ // v's distinct ID within its cohort
repeat
  if $cID = 1$ **then**
    v broadcasts on root's channel
  else v listens on root's channel
  if there was no collision **then**
    the lone broadcaster is the leader
  else let $\ell_{\max} = cNodes$'s level
    $\ell \leftarrow$ SPLITSEARCH$(0, \ell_{\max}, cSize, cNode, cID)$
    if $cID = 1$ **then**
      v broadcasts on $a_{\ell-1}(v)$'s channel
    else v listens on $a_{\ell-1}(v)$'s channel
    if collision at $a_{\ell-1}(v)$ **then**
      if v in right subtree of $a_{\ell-1}(v)$ **then**
        $cID \leftarrow cID + cSize$
      $cSize \leftarrow 2cSize$
      $cNode \leftarrow a_{\ell-1}(v)$
    else v becomes inactive
until leader declared

Figure 5.3: The LEAFELECTION algorithm. The SPLITSEARCH subroutine is displayed in Figure 5.4. The notation $a_\ell(v)$ refers to v's level-ℓ (i.e., depth-ℓ) ancestor in the tree of channels.

SPLITSEARCH($\ell_{\min}, \ell_{\max}, cSize, cNode, cID$)
(for active process v, and return level closest to root
where all subtrees have ≤ 1 cohort)

if $\ell_{\max} = \ell_{\min} + 1$ **then return** $\ell_{\max}$
else let $probedist = \lceil (\ell_{\max} - \ell_{\min})/cSize \rceil$
  let k be smallest value such that
  $\ell_{\min} + k \cdot probedist \geq \ell_{\max}$
  for $i < k$, define ℓ_i as $\ell_i = \ell_{\min} + i \cdot probedist$
  define $\ell_k = \ell_{\max}$
  if $cID < k$ **then**
    CHECKLEVEL(ℓ_{cID})
    CHECKLEVEL(ℓ_{cID+1})
  else do nothing for 4 rounds
  if $cID = 1$ and the first check returned "no collision" **then**
    announce 0 on channel $cNode$ and set $i \leftarrow 0$
  else if $cID < k$ and only the first check returned "collision" **then**
    announce cID on channel $cNode$ and set $i \leftarrow cID$
  else listen to $cNode$ and set i to the announced value
  return SPLITSEARCH($\ell_i, \ell_{i+1}, cSize, cNode, cID$)

CHECKLEVEL(ℓ) for active process v

broadcast on $a_\ell(v)$
if a collision occurred on $a_\ell(v)$ **then**
  broadcast on the channel for row ℓ
else listen on the channel for row ℓ
if the channel for row ℓ was silent **then return** "no collision"
else return "collision"

Figure 5.4: The SPLITSEARCH algorithm.

from a known ID space the same size as the cohort. Eventually, only one cohort remains, and the distinct IDs can be used to identify the leader for the whole network. The main point of the cohorts is to accelerate the binary searches used to identify key levels in the channel tree.

More precisely, the main algorithm LEAFELECTION consists of a sequence of iterations or *phases*. Initially, all active processes belong to their own cohorts of size 1. We associate with each cohort a distinct tree node $cNode$, which is the least common ancestor of all active processes in the cohort. Initially, therefore, the cohorts are associated with the leaves of the channel tree. The algorithm maintains the invariant that at the start of the i^{th} phase all active cohorts have exactly 2^{i-1} active processes, and each active process in a particular cohort has a distinct cID (or cohort ID) from $[2^{i-1}]$. We call the process with $cID = 1$ in the cohort the *cohort master*.

Each phase begins by testing whether more than one cohort exists by having the cohort masters broadcast on the root channel. If there is more than one cohort, then the phase must guarantee that (1) at least one cohort exists at the end of the phase, and (2) all cohorts become twice as large. We achieve this goal by pairing together some cohorts and having others become inactive. In particular, we employ SPLITSEARCH to identify the level ℓ closest to the root such that all cohorts have a different level-ℓ ancestor. Since we choose the smallest such level, there exists at least one pair (and possibly many pairs) of cohorts that have the common level-$(\ell - 1)$ ancestor. To identify these pairs, each cohort master broadcasts on the channel for its level-$(\ell - 1)$ ancestor (and all other members of the cohort listen). If there is a collision, then this cohort can be paired; otherwise, it becomes inactive. To pair the cohorts, we observe that one cohort is in the ancestor's left subtree, whereas the other must be in the right subtree, so we increase the $cIDs$ of processes in the right subtree by the cohort size. Finally, the cohort tree node $cNode$ can be updated to this ancestor.

The SPLITSEARCH routine is similar to SPLITCHECK in Section 5.2, with the exception of two key differences. First, the paths down to active cohorts may diverge at different points, but we wish to identify the smallest level ℓ globally where all have diverged. We thus employ a slightly more complicated test, called CHECKLEVEL to test whether any nodes share an ancestor at level ℓ. The test consists of two rounds. During the first round, one node per cohort broadcasts on its level-ℓ ancestor. Some processes may observe a collision, whereas others may not. To arrive at the same conclusion, any collisions are announced on the row channel for level-ℓ. The second key difference in SPLITSEARCH is that we exploit the large size of cohorts to accelerate the search. In particular, let p be the size of cohorts. Then the search is a $(p+1)$-ary search, instead of a binary search, adapted from Snir's [98] CREW parallel search. The search takes as input a range of levels $(\ell_{\min}, \ell_{\max}]$ to search. This range is then subdivided into $p+1$ subranges of roughly the same size, given by $(\ell_0 = \ell_{\min}, \ell_1], (\ell_1, \ell_2], (\ell_2, \ell_3], \ldots, (\ell_{k-1}, \ell_k = \ell_{\max}]$. (Usually $k = p+1$, except within one recursion of the base case where k can be smaller.) The observation is that each subrange can be tested independently—if there is a collision at level ℓ_i but no collision at level ℓ_{i+1}, then $(\ell_i, \ell_{i+1}]$ is the subrange to search. Thus, we can test the subranges in parallel by assigning one process per cohort to each range according to cIDs. Once the correct subrange has been identified by one process per cohort, that process announces the range to its comrades on the cohort tree node $cNode$. Note that this step requires listeners to read and interpret the message, not just listen for collisions. In this way, all processes know which subrange to recurse in, and the size of the range has been reduced by roughly a $p+1$ factor.

ANALYSIS

To prove correctness, we will argue that the following structural properties are invariant across phases. Assuming the properties hold for each iteration, we first prove that subrou-

tines operate correctly (Lemmas 5.3.9 and 5.3.10). We then inductively argue that Property 5.3.8 indeed holds with Lemma 5.3.11.

Property 5.3.8. *Let i be the phase number of* LEAFELECTION*:*

- *All active processes belong to a cohort.*
- *Each cohort has exactly $cSize = 2^{i-1}$ active processes as members.*
- *Each member of a particular cohort has a distinct identifier $cID \in [2^{i-1}]$.*
- *The cohort process is the tree node that is the least common ancestor of all members (which correspond to leaves). All cohort processes occur at distinct tree node in the same level. Moreover, all members of a cohort correctly identify their cohort process with $cNode$.*

Lemma 5.3.9. *Suppose that Property 5.3.8 holds and that exactly one process in each active cohort performs* CHECKLEVEL(ℓ)*. Then the return value is "no collision" if and only if all cohorts have distinct level-ℓ ancestors.*

Proof. By assumption, all cohort processes occur at the same level. If ℓ is a descendent level (i.e., ℓ is at least the level of cohort processes), then CHECKLEVEL correctly returns "no collision"—by Property 5.3.8, each participating process is a descendent of a distinct cohort process, so they broadcast on different channels.

Suppose instead that ℓ is smaller than the cohort processes' level. If two cohorts share an ancestor at level ℓ, then by Property 5.3.8 all members of those cohorts also share that ancestor. Thus, the first broadcast observes a collision, which is advertised to all other cohorts in the second broadcast. If no cohorts share an ancestor at that level, then no collision is observed by anyone. □

Lemma 5.3.10. *Suppose that Property 5.3.8 holds when* SPLITSEARCH *is called from* LEAFELECTION*. Then* SPLITSEARCH *correctly returns the smallest level ℓ (nearest to root) such that all cohorts have distinct level-ℓ ancestors.*

Proof. The proof is by induction over the recursive calls. The hypothesis is that on each call (1) all concurrent calls are synchronized and use the same values of $\ell_{\min}$ and $\ell_{\max}$, (2) $\ell_{\min} < \ell$, and (3) $\ell \leq \ell_{\max}$. Assuming the hypothesis, the search only terminates if the correct value is identified. It is easy to verify that the search eventually terminates because the range gets strictly smaller on each recursive call.

For the base case, (1) follows from the fact that all cohort processes have the same level, and hence all active processes agree on $\ell_{\max}$. (3) is true trivially since all cohort processes are distinct tree nodes at level $\ell_{\max}$. Since the search only executes if there was a collision at the root, we know $0 < \ell$ and hence (2) holds.

For the inductive step, it is easy to verify that $\ell_{\min} = \ell_0 < \ell_1 < \cdots \ell_k = \ell_{\max}$. Moreover, due to distinct cIDs (Property 5.3.8), CHECKLEVEL(ℓ_i) is performed by exactly one process in each cohort at a time. Thus, by Lemma 5.3.9 these calls perform the correct answers, with the corresponding processes in each cohort observing the same answers. Moreover, CHECKLEVEL(ℓ_i) returns "collision" for all $\ell_i < \ell$ and "no collision" for $\ell_i \geq \ell$, so only one subrange can be identified for the recursive search and hence only one process per cohort makes an announcement. (This is essentially the same argument as Section 5.2 as well as the parallel search [98].) Finally, we must verify that the range is announced without collision and that all processes in the cohort listen, which follows from the assumption that each cohort has a different $cNode$. □

Lemma 5.3.11. *Property 5.3.8 holds at the start of the i^{th} phase of* LEAFELECTION.

Proof. The proof is by induction. By assumption that each active process is a separate leaf of the tree, the property holds trivially initially with all cohorts consisting of a single active process.

Suppose the property holds at the start of the ith phase. Then we must show that it holds at the start of the next iteration. Since SPLITSEARCH returns the correct answer

(Lemma 5.3.10), we know that (1) all cohorts have distinct level-ℓ ancestors, (2) at least one level-$(\ell - 1)$ tree node has multiple descendent cohorts, and (3) all such level-$(\ell - 1)$ tree nodes have exactly two descendent cohorts, one in the left and one in the right subtree. Since exactly one process per cohort broadcasts on its level-$(\ell - 1)$ ancestor, the cohorts observe a collision if and only and only if they share a level-$(\ell - 1)$ ancestor with another cohort. Thus, cohorts remain active if and only if they can be paired with another cohort, thereby doubling the size of the cohort. Since the cIDs for a cohort are by inductive assumption distinct values from $[cSize]$, adding $cSize$ to the IDs of one of the paired cohorts preserves distinctness. Moreover, the new $cNode$ is indeed the least common ancestor of both cohorts, and hence of all processes therein. □

We are now ready to bound the performance of LEAFELECTION. We first bound the number of phases as a direct corollary of Property 5.3.8. Then we bound the cost of the searches.

Corollary 5.3.12. *If* LEAFELECTION *begins with x active processes assigned to distinct leaves of the channel tree, then it correctly identifies a leader in $O(\log x)$ phases.*

Proof. By Lemma 5.3.11, Property 5.3.8 holds, and hence at the start of the ith phase all cohorts have size 2^{i-1}. Thus, there cannot be more than $(\lg x + 1)$ phases. □

Lemma 5.3.13. *During the ith phase,* SPLITSEARCH *completes in $O((1/i) \log h)$ rounds, where $h = \lg \mathcal{C}$ is the height of the channel tree.*

Proof. With each recursive call, the size of the level subrange to search becomes $\left\lceil \frac{\ell_{\max} - \ell_{\min}}{cSize} \right\rceil$, i.e., reducing by at least a $\Theta(cSize) = \Theta(2^i) + 1$ factor. Thus, the number of recursive calls is $O(\log_{2^i+1} h) = O(\log h / \log 2^i) = O((1/i) \log h)$. Noting that each call is a constant number (specifically 5) of rounds completes the proof. □

Finally, we have our main theorem for this step of leader election:

Theorem 5.3.14. *If* LEAFELECTION *begins with x active processes assigned to distinct leaves of the channel tree, then it correctly identifies a leader in $O(\log h \log\log x)$ phases, where $h = \lg \mathcal{C}$ is the height of the channel tree.*

Proof. Correctness follows from the Lemma 5.3.11. To prove the performance, we observe that the work of each of the $\lg x$ phases is dominated by the SPLITSEARCH. Applying Corollary 5.3.12 and Lemma 5.3.13, we conclude that the total number of rounds is

$$\sum_{i=1}^{O(\log x)} O((1/i)\log h) = O\left(\log h \sum_{i=1}^{O(\log x)} (1/i)\right) = O(\log h \log\log x).$$

□

Recall that if the previous steps are successful, then $x = O(\log n)$, and hence this bound reduces to:

$$O(\log\log \mathcal{C} \cdot \log\log\log n) = O(\log\log n \cdot \log\log\log n).$$

5.4 FAIRNESS OF CONTENTION RESOLUTION ALGORITHMS

In this section, we give a formal definition of fairness for contention resolution algorithms. Our intuition is that a fair contention resolution algorithm should provide each active process roughly equal chance to be selected as the leader (i.e., the process that first transmits alone to solve the problem). We will then prove that both TWONODE and SPLITELECT are fair due to the uniform randomness during the ID reduction stages.

We begin, as in Chapter 3, by defining the inequality factor of contention resolution algorithm $\mathcal{A}$ with respect to a maximum ratio of the relevant probabilities:

Definition 5.4.1. *Fix a contention resolution algorithm $\mathcal{A}$ and a set $A \subseteq V$ of active processes. Define $p(i)$ to be the probability that process $i \in A$ is the first process to transmit alone on channel 1 in an execution of $\mathcal{A}$ with active set A. The inequality factor of algorithm $\mathcal{A}$ with respect to active set A is defined as*

$$F_{\mathcal{A},A} = \max_{i,j \in A} \left\{ \frac{p(i)}{p(j)} \right\}.$$

If there exists an active set A and $i \in A$ such that $p(i) = 0$, then we fix $F_{\mathcal{A},A} = \infty$.

We now leverage the notion of inequality factors to provide two definitions of fairness: one for the general case and one for the restricted case where $|A| = 2$:

Definition 5.4.2 (General Case Fairness)**.** *We say that contention resolution algorithm $\mathcal{A}$ is fair in the general case if there exists a constant $c > 0$ s.t. for every active set A with $|A| \geq 2$, $F_{\mathcal{A},A} \leq c$.*

Definition 5.4.3 (Restricted Case Fairness)**.** *We say that contention resolution algorithm $\mathcal{A}$ is fair in the restricted case if there exists a constant $c > 0$ s.t. for every active set A with $|A| = 2$, $F_{\mathcal{A},A} \leq c$.*

We now prove that TWOACTIVE is fair in the restricted case:

Theorem 5.4.4. TWOACTIVE *is fair in the restricted case.*

Proof. We are going to prove that for any $A \subseteq V$ such that $|A| = 2$, $F_{\text{TWOACTIVE},A} = 1$. Note that our TWOACTIVE algorithm is symmetric: the execution of the algorithm does not differ between different processes. In more detail, Step #2 is deterministic since the leaf node on the left always becomes the leader. Thus, the inequality factor only depends on Step #1, the ID Reduction stage. In this stage, processes choose IDs from $[1, \mathcal{C}]$ with uniform randomness, so each process has the same probability to select a smaller ID and then acts as the leaf node on the left and therefore becomes the leader. In other word, $F_{\text{TWOACTIVE},A} = 1$ for any $A \subseteq V$ with $|A| = 2$. □

Now we prove that SPLITELECT satisfies general case fairness defined in Definition 5.4.2:

Theorem 5.4.5. SPLITELECT *is fair in the general case.*

Proof. We are going to prove that for any $A \subseteq V$ such that $|A| \geq 2$, $F_{\text{SPLITELECT},A} = 1$. In fact, this is valid as long as we prove that for all $i \in A$, we have $p(i) = 1/|A|$. Note that the ID reduction of REDUCE in Step #1 is symmetric. IDREDUCTION in Step #2 is done with uniform randomness such that each process in A has the same probability $(1/|A|)$ to become the master of the final cohort. On the other hand, Step #3 deterministically chooses the master of the final cohort as the leader. As a result, $F_{\text{SPLITELECT},A} = 1$. □

CHAPTER 6

FAIR ASYNCHRONOUS SHARED MEMORY ALGORITHMS

In the last part of this thesis, we turn our attention to the asynchronous shared memory model. We will study the novel notion of fair consensus in this setting.

The consensus problem is one of the most popular topics in the study of distributed computation [79]. The consensus problem assumes each process starts with an initial value. To solve the problem, an algorithm must terminate with every process agreeing on the same output value chosen from the initial values. An important application of consensus is to maintain consistency in distributed fault-tolerant systems. Research on consensus commonly assumes crash failures [94], a fault that causes a process to stop taking steps. It has been proved that consensus is impossible even with one single failing process in the read/write shared memory model [30, 79]. To make consensus achievable, later work focuses on adding additional assumption to the basic model. For example, Paxos [66] assumes leader election oracles in the system, and Raft [84] adds even stronger assumptions about leaders as well as the availability of log management primitives.

This chapter will study a novel property of consensus algorithms: their fairness. A *fair* consensus algorithm, intuitively, guarantees that every initial value is decided with similar probability. We will formalize definitions of fairness and explore assumptions under which fair consensus is achievable. Next, as an application of these new algorithms, we will use them to implement a replicated state machine [61]. Replicated state machines can be used in the construction of a fault-tolerant client-server system, as well as to provide synchronization to an asynchronous system [96]. We show that the use of fair consensus algorithms

simplifies the implementations of state machines and provide stronger fairness properties to the applications that leverage these machines for consistency.

Note that, as mentioned above, consensus algorithms that are fault-tolerant and fair are impossible with an arbitrary scheduler in the asynchronous shared memory model as fault-tolerant consensus itself (without fairness) is impossible. To overcome this problem, in this chapter we will assume a more constrained scheduler that is still quite general and adversarial. Alistarh *et al.* [2] propose a *stochastic scheduler* that arranges operations according to stochastic distributions. Under this assumption of stochastic schedulers, any lock-free algorithm is proved to be wait-free with probability 1. We will use the stochastic scheduler to explore new consensus algorithms that are both fault-tolerant and fair.

6.1 MODEL

This chapter considers variations of the asynchronous shared memory model [79].[1] The asynchronous shared memory model assumes a collection of n (≥ 2) distributed processes, each running an instance of a distributed algorithm, and communicating through shared memory objects. We assume that each process has a unique identifier from $\{1, 2, \ldots, n\}$. Here we confine our attention to registers as the main shared memory object we are considering. For the sake of clarity, in the pseudocode that followed, shared memory variables will be highlighted by underscored variable names.

In the sections that follow, we will define the operation of this model.

6.1.1 OPERATION STEPS, EXECUTIONS AND CRASH FAILURES

There are two different types of operation steps that processes can take in the asynchronous shared memory model: local steps that only use the local memory of the process, and shared memory steps that require access to shared variables.

An execution in the asynchronous shared memory model is defined as a sequence of local steps and shared memory accesses performed by processes. To simplify definitions at only a minor cost to generality, we assume that executions are scheduled in discrete rounds (e.g., as was assumed in [2]), where in each round, a single process can take any number of local steps as well as random coin flips that generates stochastic results (for modeling purposes, we assume the result of the coin flips are taken from an infinite random bit array provided as input to the process by the system), until it gets to a shared memory operation. It then takes the shared memory step atomically to end the round. As detailed below, the

[1]Note that asynchronous message passing systems can be simulated by asynchronous systems using shared memory [79]. That is, our shared memory solutions can run in the network model as well by running simulations of the shared registers (when no more than half processes can fail at the same time during an execution).

choice of which process takes steps in a round is decided by a scheduler. Accordingly, in the following, we often refer to these rounds as "scheduler rounds".

The model also assumes crash failures. If a process crashes during a round then it permanently stops taking steps before reaching that round's shared memory step. We assume an upper bound t on the total number of failures allowed in the network, and $f \leq t$ denotes the actual number of failures in a given execution. Note that deterministic consensus is not achievable in this model for any $t \geq 1$, under the assumption of a worst-case scheduler [30]. Crashes show up as an event in the execution. The decision of whether or not a process crashes in a given round is also decided by a scheduler.

6.1.2 Stochastic Scheduler

Executions in the asynchronous shared memory model require a *scheduler* to decide which process takes a step or crashes in each round of the execution. In more detail, a scheduler produces a schedule, which is defined as a sequence of process IDs and crash events. The schedule $(i, j, (i \text{ crashes}), k)$ can be interpreted as: process i takes its steps in round 1, process j takes its steps in round 2, process i crashes in round 3, and process k takes its steps in round 4.

Before defining how a scheduler produces a schedule, we must first define the notion of a *history*: the history of round r, denoted by $H(r)$, is the schedule describing which processes took steps or crashed in the first $r-1$ rounds of the execution. In more detail, the history of round r provides the information about the first $r-1$ rounds including who took steps or crashed in each round, and in the case of a shared memory step what type of shared memory access they made (but without the values of the access). For a given history H, $K(H)$ is the set of processes that did not crash in the history.

Formally, a scheduler $\mathcal{S}$ is a pair containing a step distribution function π for scheduler round r, and a crash set C. The step distribution function π takes history H as input, and

outputs a probabilistic distribution $\Pi(r) = \{\gamma_i(r)\}_{i=1}^{n}$, where $\gamma_i(r) \geq 0$ is the probability for process i to be scheduled to take steps in the scheduler round r. Crash set C contains pairs of the processes to crash and the round during which they crash. Set C is predefined and C cannot contain more than t processes (note that t is a fixed system parameter). In more detail, a scheduler $\mathcal{S}$, defined as a step distribution function π and crash set C, provides the scheduling decision for a given round r of an execution with history $H(r)$ as follows:

1. Look at C and see if there is any process to crash in this scheduler round. If there is, schedule the corresponding crash event for scheduler round r; else, go to step 2.

2. Map history $H(r)$ to step distribution $\Pi(r)$ using function π. Then go to step 3.

3. Select a process to take steps in scheduler round r according to distribution $\Pi(r)$.

There are several restrictions on distribution $\Pi(r)$. First, for any correct process $i \in K(H(r))$, the set of correct processes by the end of scheduler round $r-1$, we have $\gamma_i(r) \geq 0$. However, if process i crashes in scheduler round r, it should satisfy that $\gamma_i(r') = 0$ for all $r' \geq r$. Moreover, the distribution must satisfy the normalization requirement $\sum_{i=1}^{n} \gamma_i(r) = 1$. We define the threshold of scheduler $\mathcal{S}$ as $\theta(\mathcal{S}) = \inf_{\substack{i \in K(H(r)) \\ r>0}} \{\gamma_i(r)\}$. That is to say, a correct process will always be scheduled to take the next step with probability at least $\theta(\mathcal{S})$.

If $\theta(\mathcal{S}) = 0$, then we say the scheduler is a worst-case scheduler, capturing the fact that it can produce arbitrary schedulers. A scheduler $\mathcal{S}$ is stochastic if $\theta(S) > 0$; that is, there is some non-zero lower bound probability such that in every scheduler round every non-crashed process has at least that probability of taking the next step. The idea of a stochastic scheduler was recently introduced by Alistarh *et al* [2] who used it to study liveness properties of distributed data structure implementations. As we will show, this minor constraint will enable us to achieve consensus algorithms that are both fault-tolerant and fair.

6.2 THE CONSENSUS PROBLEM

In this section, we define the single instance and multi-instance consensus problems we will study.

Assume that each process i is provided with an initial value v_i from some fixed value set V, where we assume $|V| > 1$. Every process is provided the ability to perform a single irrevocable decision of a value from V. Consensus problem is defined as follows:

Definition 6.2.1 (Consensus)**.** *A distributed algorithm $\mathcal{A}$ is said to solve the* consensus *problem in this setting if $\mathcal{A}$ satisfies:*

- *Validity: if $w \in V$ is decided, then there exists a process j such that $v_j = w$.*
- *Agreement: all decision values are identical.*
- *t-Failure Probabilistic Termination ($0 \leq t \leq n$): if there are at most t crashed processes during an execution α, then for all $\epsilon > 0$, there exists $Q > 0$ such that*

$$\Pr\{\textit{some correct process that has not decided by scheduler round } Q\} < \epsilon.$$

The termination property defined above is the typical property used to study randomized consensus algorithms; i.e., they are allowed to have some probability of not terminating that decreases over time. We use the probabilistic version of this property here because our algorithms will depend on the probabilistic behavior of the stochastic scheduler to guarantee termination.

Multi-instance consensus. In many real-world settings, consensus algorithms are seldom executed for only one instance. Instead, we may need to run multiple instances of consensus in a row, potentially using different initial values for different instances. We will formalize this multi-instance consensus scenario as follows.

We assume a user for each process that provides an initial value to the process at the beginning of the execution. Later, a process can return a value to this user as a local step: an action that has the user immediately respond with the initial value for the next instance. We use $v_i^{(m)}$ to indicate the m^{th} value passed by the user to process i, and $w_i^{(m)}$ describes process i's decision value. We assume that the user is well-formed in the sense that it will wait for a decision before passing down another input value. We say a multi-consensus algorithm is ***correct*** if it guarantees to satisfy the three consensus properties (validity, agreement, and t-failure probabilistic termination) with respect to the initial and decision values for each instance of consensus.

6.3 FAIR CONSENSUS DEFINITION

Now we provide a definition of fairness for consensus algorithms. For reasons we will elaborate below (see Section 6.4), we focus only on multi-instance consensus. The intuition for our definition is that each correct process should have a roughly equal chance of having its value decided in the limit. Without the loss of generality, we assume that all initial values are unique.[2]

We begin by defining the useful notions of decision frequency and inequality factors. Roughly speaking, we define the decision frequency of a process as the expected times that its initial values are decided in a fixed number of instances, and the inequality factor of a consensus algorithm as the maximum ratio of decision frequencies between correct processes. We say that a consensus algorithm is fair if its inequality factor converges to a constant as the number of instances goes to infinity. We now formalize these concepts:

Definition 6.3.1. *Given an execution α of a correct multi-instance consensus algorithm $\mathcal{A}$ and instance count $m \geq 1$, let the m-decision sequence of α, denoted by $d_{\mathcal{A}}(\alpha, m)$, be the sequence describing the first m decision values.*

Definition 6.3.2. *Given a correct multi-instance consensus algorithm $\mathcal{A}$, scheduler $\mathcal{S}$ and instance count m, the decision frequency of a fixed process i is the number of times process i's values show up in the m-decision sequence $d_{\mathcal{A}}(\alpha, m)$. We define the expected decision frequency, denoted by $f_{\mathcal{A},\mathcal{S}}(i, m)$, as the expected decision frequency of process i over all possible executions of consensus algorithm $\mathcal{A}$ running with scheduler $\mathcal{S}$ for a given instance count m.*

Definition 6.3.3. *Fix a correct multi-instance consensus algorithm $\mathcal{A}$, scheduler $\mathcal{S}$ with crash set C. Let K be the set of processes that do not show up in C. Fix some instance count*

[2]Note that this is easy to achieve by attaching process unique IDs to their initial values, for example by transforming v_i to (v_i, i).

$m \geq 0$. *Then the inequality factor of consensus algorithm $\mathcal{A}$ through m decisions under scheduler $\mathcal{S}$, denoted by $F_{\mathcal{A},\mathcal{S}}(m)$, is defined as the maximum ratio of expected decision frequencies of length m of all correct processes in K:*

$$F_{\mathcal{A},\mathcal{S}}(m) = \max_{\substack{i,j \in K \\ i \neq j}} \left\{ \frac{f_{\mathcal{A},\mathcal{S}}(i,m) + 1}{f_{\mathcal{A},\mathcal{S}}(j,m) + 1} \right\}.$$

Note we add 1 to both denominator and numerator in the definition is to exclude the possibility of a "divide-by-zero" error.

Definition 6.3.4. *We say that a multi-instance consensus algorithm $\mathcal{A}$ is fair under scheduler $\mathcal{S}$ if there exists a constant c and instance count $m_0 \geq 0$ such that for all $m > m_0$, $F_{\mathcal{A},\mathcal{S}}(m) < c$. In other word, consensus algorithm $\mathcal{A}$ is fair if the inequality factor of $\mathcal{A}$ converges to a constant as m goes to infinity.*

6.4 IMPOSSIBILITY OF FAIR SINGLE-INSTANCE CONSENSUS

In this section, we motivate our decision to focus primarily on fairness for multi-instance consensus, by formalizing the intuitive result that fairness is impossible to achieve for single instance consensus, even with a stochastic scheduler.

Theorem 6.4.1. *Fix a single instance consensus algorithm $\mathcal{A}$ that satisfies 1-failure probabilistic termination. There exists a stochastic scheduler $\mathcal{S}^*$ such that $F_{\mathcal{A},\mathcal{S}^*}(1) = \Omega(n)$.*

Before giving the proof of this theorem, we are going to prove that a process must be scheduled at least once before its initial value is decided.

Lemma 6.4.2. *Fix a correct single instance consensus algorithm $\mathcal{A}$ and stochastic scheduler $\mathcal{S}$. If process i's initial value v_i is decided by scheduler round r, then process i was scheduled at least once in the first r scheduler rounds.*

Proof. Assume for contradiction that process i's initial value v_i is decided by some process j during scheduler round r, even though process i took no step. Because the scheduler does not know processes' initial values, the probability that this same execution occurs if we start i with a different unique initial value, for example v_i', is the same. Therefore, there is a non-zero probability of a validity violation, which leads to a contradiction to the assumption that $\mathcal{A}$ is correct. □

We now prove our main theorem.

Proof (of Theorem 6.4.1). First consider the scheduler $\mathcal{S}$ parameterized by:

- $\Pi(r) = \{\gamma_i(r)\}_{i=1}^n : \gamma_i(r) = \begin{cases} 0 & i = 1 \\ \dfrac{1}{n-1} & \text{otherwise} \end{cases} \quad \forall r \geq 1;$
- $C = \{(1, 1)\}$.

In other word, in any execution scheduled by $\mathcal{S}$, process 1 will crash from the very first round and all the other processes have equal probability to be scheduled in each round. By the definition of 1-failure probabilistic termination, we know that there exists an $R > 0$ such that

$$\Pr\{\mathcal{A} \text{ decides a value by scheduler round } R \text{ under } \mathcal{S}\} \geq 1 - \frac{1}{n^2}.$$

Now consider another scheduler $\mathcal{S}^*$ parameterized by

- $\Pi(r) = \{\gamma_i(r)\}_{i=1}^n : \gamma_i(r) = \begin{cases} \frac{1}{n^2 R} & i = 1 \\ \frac{1 - \frac{1}{n^2 R}}{n-1} & \text{otherwise} \end{cases} \quad \forall r \geq 1;$
- $C = \emptyset$.

Like $\mathcal{S}$, $\mathcal{S}^*$ is also a stochastic scheduler. We are going to prove that single instance consensus algorithm $\mathcal{A}$ cannot be fair under stochastic scheduler $\mathcal{S}^*$. In particular, we will show that $f_{\mathcal{A},\mathcal{S}^*}(1,1)$, the decision frequency (probability) of process 1, will have a difference in $\Omega(n)$ as compared to other processes.

Note that for any process other than process 1, scheduler $\mathcal{S}^*$ is almost the same as $\mathcal{S}$, and any execution α^* under $\mathcal{S}^*$ is almost the same as a certain execution α under $\mathcal{S}$. In more detail, if we take a union bound on the probability for process 1 to be scheduled in the first R rounds, then

$$\Pr\{\text{process } 1 \text{ is scheduled by } \mathcal{S}^* \text{ in the first } R \text{ rounds}\} < \frac{R}{n^2 R} = \frac{1}{n^2}.$$

We are going to prove that $\mathcal{A}$ still has a high probability of deciding a value by round R even if we switch the scheduler from $\mathcal{S}$ to $\mathcal{S}^*$. To make our notation concise, we define E_1 to be the event "$\mathcal{A}$ decides a value by round R under $\mathcal{S}^*$" and E_2 to be the event "process 1 is not scheduled in the first R rounds by $\mathcal{S}^*$". Then

$$\Pr(\bar{E}_2) = \Pr\{\text{process } 1 \text{ is scheduled by } \mathcal{S}^* \text{ in the first } R \text{ rounds}\} < \frac{1}{n^2}$$

and

$$\begin{aligned}
\Pr(\bar{E}_1) &= \Pr(\bar{E}_1|E_2)\Pr(E_2) + \Pr(\bar{E}_1|\bar{E}_2)\Pr(\bar{E}_2) \\
&< \Pr\{\mathcal{A} \text{ does not decide a value by round } R \text{ under } \mathcal{S}^*|E_2\} \cdot 1 + 1 \cdot \frac{1}{n^2} \\
&= \Pr\{\mathcal{A} \text{ does not decide a value by round } R \text{ under } \mathcal{S}\} + \frac{1}{n^2} \\
&\leq \frac{1}{n^2} + \frac{1}{n^2} = \frac{2}{n^2}.
\end{aligned}$$

Finally, we are going to prove that some process other than process 1 has a good chance of having its value decided under scheduler $\mathcal{S}^*$. We will start with the calculation of the probability of $\bar{E}_3$, where E_3 refers to the event that "v_1 is decided by $\mathcal{A}$ under $\mathcal{S}^*$" In fact, we have

$$\begin{aligned}
\Pr(\bar{E}_3) &\geq \Pr(\bar{E}_3|E_2)\Pr(E_2) \\
&\geq \Pr(\bar{E}_3|E_1 \cap E_2)\Pr(E_1)\Pr(E_2).
\end{aligned}$$

At the same time, Lemma 6.4.2 shows that if $\mathcal{A}$ decides a value before round R but process 1 fails to take a step in the first R rounds, then process 1 has zero probability to get its initial value decided. In other words, $\Pr(E_3|E_1 \cap E_2) = 0$ and therefore:

$$\Pr(\bar{E}_3) \geq (1 - \Pr(E_3|E_1 \cap E_2))\Pr(E_1)\Pr(E_2) = \Pr(E_1)\Pr(E_2).$$

Since we have proved that $\Pr(E_1) \geq 1 - \frac{2}{n^2}$ and $\Pr(E_2) > 1 - \frac{1}{n^2}$, we will get $\Pr(\bar{E}_3) > 1 - \frac{3}{n^2}$ and $f_{\mathcal{A},\mathcal{S}^*}(1,1) = \Pr(E_3) < \frac{3}{n^2}$

In contrast, there must be at least one of the $n-1$ other processes that has an expected decision frequency, i.e., the probability for its value to get decided, of at least $\frac{1-\frac{2}{n^2}}{n-1}$. Therefore, the inequality factor of $\mathcal{A}$ under scheduler $\mathcal{S}^*$ is greater than

$$\frac{(1-\frac{2}{n^2})/(n-1)}{\frac{2}{n^2}} > \frac{n}{3},$$

as claimed. □

6.5 FAIR CONSENSUS ALGORITHMS

As mentioned, it is impossible to solve deterministic consensus even with only a single failure. Existing solutions therefore leverage additional model assumptions, often captured as oracle formalisms. Paxos [66], for example, is constructed based on a leader election oracle (usually denoted by Ω). We would like, however, to avoid defining these assumptions formally in our model, as we want our approach to introducing fairness to apply to many different settings. We will use instead a black box approach to construct a fair consensus algorithm. In more detail, suppose BLACKBOXCONSENSUS is a consensus algorithm that potentially uses some extra assumptions or oracles to solve the problem, and satisfies validity, agreement and t-failure probabilistic termination. We will show how to create a fair consensus algorithm FAIRCONSENSUS that treats BLACKBOXCONSENSUS as a black box (we can provide initial values to BLACKBOXCONSENSUS and it will output a decision value eventually). Our transformation of BLACKBOXCONSENSUS into a fair consensus algorithm FAIRCONSENSUS will itself depend on is the stochastic scheduler defined above. For the sake of completeness, this chapter will also explore consensus solutions that rely only on the stochastic scheduler assumption. If these new consensus algorithms are used as the black box, then the result is a fair consensus algorithm that only depends on a stochastic scheduler.

6.5.1 FAIRCONSENSUS

In this part, we are going to take a multi-instance consensus algorithm BLACKBOXCONSENSUS, treated as a black box that accepts initial values and returns decisions, and transform it into a multi-instance algorithm FAIRCONSENSUS that satisfies the definition of fairness stated in Definition 6.3.4. This transformation will make use of the stochastic scheduler.

In more detail, each consensus instance of algorithm FAIRCONSENSUS begins with a call to a subroutine called COLLECT to collect initial values of all non-crashed processes identified by DETECT, a failure detection process that uses the stochastic scheduler assumptions and runs in parallel. Then each process picks one of the initial values collected as its new initial value for BLACKBOXCONSENSUS. Once we get to a point where the values from all correct processes are consistently collected, this strategy will provide good fairness guarantees.

FAILURE DETECTION

If our strategy is to collect initial values before running an instance of consensus, we need some way of knowing which processes to wait for and which to ignore because they are crashed. Here we introduce DETECT (see Figure 6.1), a failure detector subroutine, to help with this. The idea is that each process can query the failure detector process about the set of correct process IDs. We assume that the failure detector process runs in parallel with the thread where the consensus algorithm is executed.

The idea behind DETECT is simple. Initially, A_i, the set of correct processes maintained by process i, contains all processes. There is a global-time array $\underline{time}[1 \ldots n]$ in the shared memory. A correct process i will keep increasing $\underline{time}[i]$ by 1. It also keeps a local-time array $t_i[1 \ldots n]$, a local snapshot of $\underline{time}[j]$ for every process j. Process i starts reading $\underline{time}[j]$ from the shared memory after incrementing its own global-time r_i times. Whenever process i reads $\underline{time}[j]$ and fails to find an increase compared to the local copy, it assumes that process j is crashed and remove it from A_i. On the other hand, if it sees an increase in $\underline{time}[j]$ for a process j that is not included in A_i, it indicates that process j is missed out from the previous round. Then j is added back to A_i again, and the waiting time r_i will be increased by 1 at the end of this iteration. The general idea is that once r_i is incremented enough times, A_i will be able to observe an increment of the global-time of each correct

```
DETECT (for correct process i)
A_i is accessible by other threads of process i
time[1...n] is an array in the shared memory
A_i ← [1, n], r_i ← 1
t_i[1...n] ← {0, ..., 0}

do
  repeat r_i times:
    t_i[i] ← t_i[i] + 1
    write t_i[i] to time[i]

  miss_i ← FALSE
  for j ∈ [1, n] and j ≠ i do
    temp_i[i] ← read time[j]
    t_i[i] ← t_i[i] + 1
    write t_i[i] to time[i]
    if temp_i[j] ≠ t_i[j] then
      if j ∉ A_i then
        A_i ← A_i ∪ {j}
        miss_i ← TRUE
    else
      A_i ← A_i \ {j}
    t_i[j] ← temp_i[j]
  if miss_i = TRUE then
    r_i ← r_i + 1
```

Figure 6.1: The algorithm of DETECT.

process during the r_i waiting steps. The challenge is figuring out how long of an r_i value is "long enough". We will answer this question below by analyzing the threshold for these r_i values that ensures DETECT behaves correctly for a long time with high probability. Notice, however, for the sake of clarity, in the sections that later use our DETECT subroutine, we treat it simply as a black box that eventually works.

We now show that when the wait count r_i for a given process i grows sufficiently large, DETECT will maintain the actual set of correct processes in A_i for a long time with high probability. In the following, we consider one complete iteration of the main **do** loop in DETECT as an "instance" of DETECT. We number these instances $1, 2, 3, \ldots$ We define the set A_i returned by instance j of DETECT to be the value of A_i after the final step of instance j.

Lemma 6.5.1. *Fix stochastic scheduler $\mathcal{S}$ with threshold θ, a correct process i, failure probability $p > 0$, and instance count $T > 0$. Assume that at the beginning of some instance of* DETECT *run by a process i, all faulty processes have crashed and $r_i > 2\theta^{-1}(\log n + \log T + \log 2 - \log p)$. It follows that for the next T instances,* DETECT *will return a set A_i that is exactly the set of correct processes with probability at least $1 - p$.*

Proof. From Figure 6.1 we know that a process increments its global-time once every 2 shared memory steps. A process j fails to be captured in A_i if it fails to take 2 shared memory steps before process i performs r_i writes to the shared memory. We first note that for an $R = \theta^{-1}(\log(2nT) - \log p)$, the probability that process j fails to take one shared memory step in R scheduler rounds is no greater than

$$(1-\theta)^R < e^{-R\theta} = \frac{p}{2nT}.$$

The inequality above leverages the fact that $\log(1-x) \leq -x$ for $0 \leq x \leq 1$. Then the probability that j fails to take 2 shared memory steps during $2R$ scheduler rounds is no greater than $\frac{p}{nT}$. Taking a union bound on the number of processes, the probability that at least one correct process fails to take 2 steps during one iteration consisting of $2R$ scheduler rounds is less than p/T. If we take another union bound on the number of iterations, we will see that the probability that at least one process fails to take 2 shared memory steps during at least one of T iterations is less than p. In other word, A_i is correct during these T iterations after r_i reaches $2\theta^{-1}(\log n + \log T + \log 2 - \log p)$ will be at least $1 - p$. □

Eventual correct point. We proved that for any given duration T, there is a value such that once DETECT's waiting counter r_i exceeds this value, it will return the right set of correct processes for the next T updates to that set, with high probability. In the following, we analyze algorithms that use DETECT as a subroutine. To simplify these analyses, we will treat DETECT as a black box that eventually works correctly. Our actual implementation of DETECT as presented and analyzed above approximates this behavior for long periods with high probability. In more detail, in the following we assume an ideal *eventual correct point* in an execution after which no crash failure occurs and DETECT always return the right set of correct processes.

COLLECT

Here we describe and analyze the subroutine COLLECT (see Figure 6.2). This subroutine is passed a value and an instance number. Its goal is to collect all initial values for the given instance. The process running COLLECT for a given instance can then choose one of these values randomly to pass to BLACKBOXCONSENSUS.

For each instance of consensus, we will run a separate instance of COLLECT. In more detail, we assume a value array $\underline{val^{(m)}}[1 \dots n]$ for instance m in the shared memory. Process i in instance m_i with initial value $v_i^{(m_i)}$ writes its own initial value $v_i^{(m_i)}$ to $\underline{val^{(m_i)}}[i]$. Then it will keep updating A_i, the set of correct processes, from the failure detector while reading initial values from the shared memory, until initial values of all processes in A_i are collected. Clearly, it is possible to be stuck in COLLECT until the failure detector stabilizes to behave correctly, but given our above assumption of eventual correctness, this will eventually happen.

COLLECT($m_i, v_i^{(m_i)}$)
(for process i and instance m_i with initial value $v_i^{(m_i)}$)

Initialize $value^{(m_i)}[j]$ by $\perp$ for $\forall j \in [1, n]$

$C_i \leftarrow \emptyset$
write $v_i^{(m_i)}$ to $\underline{val^{(m_i)}}[i]$
do
  get A_i from DETECT
  for all $j \in A_i$ **do**
    if $value^{(m_i)}[j] = \perp$ **then**
      $value^{(m_i)}[j] \leftarrow$ **read** $\underline{val^{(m_i)}}[j]$
      if $value^{(m_i)}[j] \neq \perp$ **then**
        $C_i \leftarrow C_i \cup \{value^{(m_i)}[j]\}$
until $\forall j \in A_i$, $value^{(m_i)}[j] \in C_i$
return C_i

Figure 6.2: The algorithm of COLLECT

FAIRCONSENSUS

We are now ready to describe and analyze FAIRCONSENSUS: our fair multi-instance consensus algorithm that uses COLLECT and some BALCKBOXCONSENSUS implementation as subroutines. Suppose we are given a (possibly unfair) consensus algorithm BLACKBOXCONSENSUS that takes process ID and an initial value as input, and returns a decision value. We assume that BLACKBOXCONSENSUS satisfies validity, agreement and t-failure probabilistic termination. It follows from the termination property, that for any failure probability $p > 0$, there exists a bound $T_{\text{CONSENSUS}}(p)$, such that BLACKBOXCONSENSUS guarantees that all correct processes will decide within $T_{\text{CONSENSUS}}(p)$ scheduler rounds, with probability at least $1 - p$. As it is shown in Figure 6.3, processes run FAIRCONSENSUS for a given instance by first calling COLLECT to collect initial values from correct processes, and then choose one with uniform randomness as its new initial value for BLACKBOX-

FAIRCONSENSUS($i, m, v_i^{(m)}$)
(for process i, instance number m and initial value $v_i^{(m)}$)

$C_i \leftarrow$ COLLECT($i, m, v_i^{(m)}$)
select one value in C_i as $\tilde{v}_i^{(m)}$ with uniform randomness
$w \leftarrow$BLACKBOXCONSENSUS($i, \tilde{v}_i$)
return w

Figure 6.3: The algorithm of FAIRCONSENSUS.

CONSENSUS. The decision value of BLACKBOXCONSENSUS is also the decision value of FAIRCONSENSUS for this instance.

The analysis that follows divides the execution of FAIRCONSENSUS into three phases: *unfair phase*, *synchronization phase* and *fair phase*. The unfair phase starts from the beginning and ends after the eventual correctness point defined in the discussion of the failure detector. At this point, some "faster" correct processes might be running FAIRCONSENSUS for a larger instance number than some "slower" processes. The processes remain in the synchronization phase until the difference between these instance numbers in the system is at most 1 (that is, if correct process i is running FAIRCONSENSUS with instance value m_i, and j is running it with m_j, then $|m_i - m_j| \leq 1$). Once the instance numbers converge, the execution enters the fair phase. It is straightforward to see that once processes are in the fair phase, they never again leave it (the gap between instance numbers cannot grow beyond 1).

The validity and agreement properties of FAIRCONSENSUS are a straightforward consequence of the validity and agreement guarantees of BLACKBOXCONSENSUS.

Lemma 6.5.2 (Validity)**.** *If* FAIRCONSENSUS *decides w for instance m, then there exists $j \in [1, n]$ such that $v_j^{(m)} = w$.*

Proof. Assume for contradiction that w is not the initial value of any process. We know that COLLECT only collects "actual" initial values for that instance from processes, so w cannot be collected and therefore it will not be chosen as the initial value of any process for BLACKBOXCONSENSUS. According to the validity assumption of BLACKBOXCONSENSUS, w is not possible to be decided, which contradicts our assumption at the beginning. □

Lemma 6.5.3 (Agreement)**.** *Assume that process i calls* FAIRCONSENSUS*(m, v_i) which returns decision value w_i, and process j calls* FAIRCONSENSUS*(m, v_j) which returns w_j, then $w_i = w_j$.*

Proof. This follows from the agreement assumption of BLACKBOXCONSENSUS. □

Now we are going to prove the termination property of FAIRCONSENSUS. Seeing the three different phases of FAIRCONSENSUS, we are going to divide the termination proof into two parts. We first fix a value M, the maximum difference of the FAIRCONSENSUS instance numbers of processes when FAIRCONSENSUS enters the synchronization phase, and calculate the time required, expressed with respect for M, for the instance numbers to converge and move the execution to the fair phase. At last, we calculate the time complexity of FAIRCONSENSUS during the fair phase.

We assumed that all the synchronization phase begins after the eventual correct point. By the definition of the eventual correct point it follows that all crash failures have occurred by the beginning of this phase. This will simplify the analysis that follows. It costs time for faster processes to wait for slower processes to catch up in the synchronization phase. We are going to estimate the number of steps the fastest correct process has to wait until all other processes catch up and then enter the fair phase.

Lemma 6.5.4 (Synchronization Phase Termination)**.** *Fix a stochastic scheduler $\mathcal{S}$ with threshold θ, a failure probability $p > 0$, and some scheduler round $r > 0$ that occurs*

after the synchronization phase begins. Suppose at the beginning of scheduler round r the largest instance number for which FAIRCONSENSUS *has been called by some correct process i, but not yet returned a value to i, is M ahead of the smallest instance number. It follows that with probability at least $1 - p$, the execution will enter the fair phase within $O\left(M\left(\frac{n}{\theta}(\ln n + \ln M - \ln p) + T_{\text{CONSENSUS}}\left(\frac{p}{2M}\right)\right)\right)$ scheduler rounds.*

Proof. We first prove an important property of stochastic scheduler that for any failure probability $p > 0$ and any $R \geq (2\ln n + \ln(2M) - \ln p)/\theta$, with probability at least $1 - \frac{p}{2n^2M}$ any correct process will be scheduled at least once during these R scheduler rounds. This can be proved by using balls in bins strategy. We throw R balls into n bins. Each scheduler round corresponds to a ball and each process is represented by a bin. Ball q is in bin i is equivalent to the event that process i is scheduled in scheduler round q.

Suppose the first $n - f$ bins correspond to correct processes, and we call them "correct bins" for convenience. Since each correct process is scheduled with probability at least θ, each ball falls into a correct bin with probability at least θ. Fix a correct bin i and a duration $R \geq (2\ln n + \ln(2M) - \ln p)/\theta$, and the probability that bin i is empty at the end of these R scheduler rounds will be:

$$\Pr\{\text{bin } i \text{ is empty by the end of scheduler round } R\} \leq (1-\theta)^R \leq \frac{p}{2n^2M}.$$

The inequality above leverages the fact that $1 - x \leq e^{-x}$ for $0 \leq x \leq 1$. Taking a union bound, with a probability of at most $\frac{p}{2nM}$ there is an empty correct bin, which means that any correct process will be scheduled at least once during these R scheduler rounds with probability at least $1 - \frac{p}{2nM}$.

We are going to prove that once all correct processes have started COLLECT for some instance in the settings stated in the lemma, all correct processes will terminate COLLECT for this instance with a set containing initial values of exactly the set of correct processes after $\frac{n}{\theta}(2\ln n + \ln(2M) - \ln p)$ scheduler rounds with probability $1 - \frac{p}{2M}$. Note that for

any correct process i at this point, A_i will be exactly the set of correct processes, and the execution of COLLECT cannot terminate to start another instance until it has an initial value for the current instance from every (and only) correct process. According to the pseudocode in Figure 6.2, the collection for one particular instance requires process i to take at most n shared memory steps. Taking a union bound on our balls in bins argument above, it follows that all correct processes will take n shared memory steps within R' scheduler rounds when $R' > \frac{n}{\theta}(2 \ln n + \ln(2M) - \ln p)$ with probability at least $1 - \frac{p}{2M}$.

Now we look at the entire synchronization phase. The worst case would be that there is only one fastest process while all the other processes are in the slowest instance. The argument above tells us that with probability at least $1 - \frac{p}{2M}$, R' scheduler rounds are required for all slowest processes to complete one instance of COLLECT and finish collecting initial values for their current instance and start running BLACKBOXCONSENSUS. By our assumption, all correct processes that have started one fixed instance of BLACKBOXCONSENSUS will terminate in $T_{\text{CONSENSUS}}\left(\frac{p}{2M}\right)$ scheduler rounds with a decision value with probability at least $1 - \frac{p}{2M}$. As a result, all these slowest processes finish this instance of FAIRCONSENSUS in $R' + T_{\text{CONSENSUS}}\left(\frac{p}{2M}\right)$ scheduler rounds with probability at least $1 - p/M$. The case that not all processes other than the fastest one are in the same slowest instance requires less scheduler rounds to finish the slowest instance.

In other words, after $R' + T_{\text{CONSENSUS}}\left(\frac{p}{2M}\right)$ scheduler rounds, all correct processes are at most $M - 1$ instances behind the fastest one, with probability at least $1 - p/M$. Then with probability at least $1 - p/M$, another $R' + T_{\text{CONSENSUS}}\left(\frac{p}{2M}\right)$ scheduler rounds are required to move all correct processes at least one more instance forward. Keep doing this, and we will see that $M\left(\frac{n}{\theta}(2 \ln n + \ln(2M) - \ln p) + T_{\text{CONSENSUS}}\left(\frac{p}{2M}\right)\right)$ scheduler rounds are sufficient for slower processes to catch up with the fastest process, with probability at least $1 - p$. $\square$

We now analyze the time complexity of FAIRCONSENSUS once processes arrive in the fair phase:

Lemma 6.5.5 (Fair Phase Termination)**.** *Fix a stochastic scheduler $\mathcal{S}$ with threshold θ. Fix a failure probability $p > 0$. Fix some correct process i and scheduler round $r > 0$ that is in the fair phase. If process i call* FAIRCONSENSUS *in scheduler round r, then with probability at least $1 - p$, process i will complete this call to* FAIRCONSENSUS *within $O\left(\frac{n}{\theta}(\ln n - \ln p) + T_{\text{CONSENSUS}}(p/4)\right)$ scheduler rounds.*

Proof. In the worst case, the fastest process is at most one instance ahead of the slowest one. which requires the fastest process to wait for the slowest one to finish the previous instance before continuing the current instance. In other words, the fastest process need to wait for two instances of COLLECT and BLACKBOXCONSENSUS. According to the proof of Lemma 6.5.4, $2n\theta^{-1}(2\ln n + \ln 4 - \ln p)$ scheduler rounds are necessary for all process to finish 2 instances of COLLECT with probability at least $1 - p/2$. On the other hand, two instances of BLACKBOXCONSENSUS finish in $2T_{\text{CONSENSUS}}(p/4)$ scheduler rounds with probability at least $1-p/2$. Then process i finishes the current instance of FAIRCONSENSUS within $O\left(n\theta^{-1}(\ln n - \ln p) + T_{\text{CONSENSUS}}(p/4)\right)$ scheduler rounds with probability at least $1 - p$. □

We now turn our attention to the fairness. The fairness of FAIRCONSENSUS is straightforward given the definition of COLLECT:

Theorem 6.5.6. FAIRCONSENSUS *is fair.*

Proof. Recall that the definition of the fairness for consensus algorithms requires us to prove that eventually all correct processes have a similar chance of having their initial value decided. To show this, we fix any instance number m_0 such that FAIRCONSENSUS is called with m_0 for the first time after the fair phase begins. All crash failures have already occurred

and the failure detector is working properly by this point, so the call to COLLECT for this instance will succeed. Since the initial values for BLACKBOXCONSENSUS are chosen by processes at random, each correct process will have equal probability to get its initial value decided, which indicates the fairness of FAIRCONSENSUS. □

6.5.2 SOLVING BINARY CONSENSUS WITH A STOCHASTIC SCHEDULER

The FAIRCONSENSUS algorithm uses a black box implementation of consensus as a subroutine. For the sake of completeness, in this section and the next we explore how to solve consensus using the same stochastic scheduler constraint assumed by FAIRCONSENSUS. We start in this section by studying LEAN: a binary consensus algorithm in this setting. In the next section, we use LEAN as a subroutine in solving consensus with more general initial value sets.

In more detail, in [4] Aspnes describes a simple algorithm called LEAN-CONSENSUS that solves consensus in the "noisy scheduler model". Here we will show that a revised version of this algorithm solves consensus with a stochastic scheduler, and bound the scheduler rounds required for termination. We rename it by LEAN, and the pseudocode is rewritten in Figure 6.4. Our LEAN algorithm is slightly different from LEAN-CONSENSUS in two aspects. First, to reduce the scheduler rounds for all processes to complete the execution of LEAN, we have faster processes write the decision value to a shared memory variable $\underline{decision}$ when deciding it. In this case, slower processes only need to take constant shared memory steps to decide a value. Second, we use our own stochastic scheduler assumptions in the analysis. In the following, as in [4], we use the term "round r" for a given process to refer to that process's iteration of the main loop of LEAN that starting by reading position r in the shared memory arrays.

It has been proved in [4] that LEAN satisfies validity and agreement. We can use these two lemmas directly since our modifications do not change the proof strategies.

LEAN (for active process i with initial value $b \in \{0, 1\}$)

Initially $\underline{a_0}[0] = \underline{a_1}[0] = 1$,
$\underline{a_0}[r] = \underline{a_1}[r] = 0$ for all $r \geq 1$
$\underline{decision} = \perp$

$r \leftarrow 1$
repeat
  $w_i \leftarrow$ **read** $\underline{decision}$
  if $w_i \neq \perp$ **then**
    decide w_i and exit

  read $\underline{a_0}[r]$ and $\underline{a_1}[r]$
  if $\underline{a_b}[r] = 0$ and $\underline{a_{1-b}}[r] = 1$ **then**
    $b \leftarrow 1 - b$
  write 1 to $\underline{a_b}[r]$
  read $\underline{a_{1-b}}[r - 1]$
  if $\underline{a_{1-b}}[r - 1] = 0$ **then**
    write b to $\underline{decision}$
    decide b and exit
  else $r \leftarrow r + 1$

Figure 6.4: The algorithm of LEAN.

Lemma 6.5.7 (Validity)**.** *If every process starts with the same input bit b, every process decides b after taking 6 shared memory steps.* □

Lemma 6.5.8 (Agreement)**.** *If some process decides b at round r, then (a) no process ever writes $a_{1-b}[r]$, and (b) every process decides b at or before round $r+1$.* □

LEAN-CONSENSUS is proved to terminates in $\Theta(\log n)$ rounds with noisy schedulers and crash failures in [4]. Here we will show it terminates in $O((c \ln n) \cdot (\theta^{-12} + n\theta^{-1}))$ scheduler rounds with at most t crash failures, where $\theta = \theta(\mathcal{S}) > 0$ is the threshold of a stochastic scheduler $\mathcal{S}$ we fix for the analysis that follows.

CRASH-FREE TERMINATION

Here we study how many scheduler rounds are needed by LEAN to decide a value with high probability, in a crash-free system ($t = 0$). We will then generalize to accommodate crashes.

In round r of LEAN, process i needs to perform at most 6 shared memory operations before terminating: read $\underline{decision}$, read $\underline{a_0}[r]$, read $\underline{a_1}[r]$, write $\underline{a_b}[r]$, read $\underline{a_{1-b}}[r-1]$, and then write $\underline{decision}$. Each round of LEAN therefore consists of 6 shared memory steps in a row. Based on this observation, we prove the following lemma:

Lemma 6.5.9. *Suppose after the scheduler selects the process to take steps in scheduler round $q \geq 0$, process i is the process that has proceeded the farthest in the algorithm (breaking ties arbitrarily). Then by scheduler round $q + 12$, process i decides a value with a probability at least θ^{12}.*

Proof. We first argue that once process i is scheduled 12 times in a row, it will finally decide a value. Suppose at some point process i is in round r. Assume without the loss of generality that process i has $b = 0$. Then $\underline{a_0}[r-1] = 1$ for process i wrote 1 to it in round $r-1$. Consider the following different cases:

- Case $\underline{a_1}[r-1] = 0$. It indicates that no one has written to $\underline{a_1}[r-1]$. If process i takes 6 shared memory steps in a row, it will write b to $\underline{decision}$, decide b and terminate.

- Case $\underline{a_1}[r-1] = 1$ and $\underline{a_1}[r] = 0$. Process i will finish round r in 5 shared memory steps without a decision value and enters round $r+1$, falling in previous case. In another 6 shared memory steps, a value will be decided.

- Case $\underline{a_1}[r-1] = 1$ and $\underline{a_1}[r] = 1$. It indicates that some process has written to $\underline{a_1}[r]$. Since process i is one of the fastest processes, it has written 1 to $\underline{a_0}[r]$. Then in 1 shared memory step it finishes round r without a decision value and enters round $r+1$, falling in previous case. Then another 11 shared memory steps are needed to decide a value.

Given the observation above, the event that fastest process takes 12 shared memory steps in a row happens with probability $\prod_{k=1}^{12} \gamma_i(q+k) \geq \theta^{12}$, where $\gamma_i(r)$ is the probability that process i is scheduled in scheduler round r. □

If we divide the execution of LEAN into blocks each consisting of 12 scheduler rounds, we know from the above lemma that the probability that the farthest process decides a value in such a block is at least θ^{12}. Suppose X_j ($j = 1, 2, \ldots$) is the random indicator variable defined below:

$$X_j = \begin{cases} 1 & \text{The farthest process in block } j-1 \text{ is scheduled in all } 12 \text{ scheduler rounds in block } j \\ 0 & \text{Otherwise} \end{cases}.$$

Define $Y_J = \sum_{j=1}^{J} X_j$. Then $Y_J \geq 1$ means that in a certain block $j' \leq J$, the farthest process in block $j'-1$ is scheduled in all 12 scheduler rounds in block j' (and a value will be therefore decided). By Lemma 6.5.9, $\Pr\{X_j = 1\} = E[X_j] \geq \theta^{12}$, and by linearity of expectation $E[Y_J] \geq \theta^{12} J$.

We want to use the following Chernoff bound to prove that $Y_J \geq 1$ holds with high probability:

$$\Pr\{X < (1-\delta)\mu\} < \exp\left\{-\frac{\delta^2\mu}{2}\right\}, \text{ where } \mu = E[X] \text{ and } 0 < \delta < 1.$$

Note that the application of Chernoff bound requires the independence of random variables, while X_j's are not obviously independent. In this case, we need a stochastic dominance analysis.

In more detail, assume that $\hat{X}_j$ is a random variable such that $\Pr\{\hat{X}_j = 1\} = \theta^{12}$ and $\Pr\{\hat{X}_j = 0\} = 1 - \theta^{12}$. Define $\hat{Y}_J = \sum_{j=1}^{J} \hat{X}_j$. Then $\hat{Y}_J$ is the sum of independent random variables and we are able to apply Chernoff bound on $\hat{Y}_J$. We now leverage these definitions and observations in the lemma that follows.

Lemma 6.5.10 (Crash-Free Termination)**.** *The execution of* LEAN *terminates in* $O((c\ln n)\cdot(\theta^{-12} + n\theta^{-1}))$ *scheduler rounds with probability at least* $1 - \frac{1}{n^c}$.

Proof. We first prove that if $J \geq 2\theta^{-12}(c\ln n + \ln 2 + 1)$ for some constant $c > 0$, then $\Pr\{Y_J = 0\} < \frac{1}{2n^c}$. Suppose $\hat{X}_j$'s and $\hat{Y}_J$ are defined as above. We say that X_j stochastically dominates $\hat{X}_j$. Then $E[\hat{Y}_J] = E[\sum_{j=1}^{J} \hat{X}_j] = \theta^{12}J$. Apply Chernoff bound on $\hat{Y}_J$ and we have

$$\Pr\{\hat{Y}_J < 1\} < \exp\{-\frac{\theta^{12}J + \frac{1}{\theta^{12}J} - 2}{2}\} < \exp\{-\frac{\theta^{12}J}{2} + 1\}.$$

Since $J \geq 2\theta^{-12}(c\ln n + \ln 2 + 1)$,

$$-\frac{\theta^{12}J}{2} + 1 \leq -c\ln n - \ln 2$$

and then

$$\Pr\{\hat{Y}_J < 1\} < \exp\{-c\ln n\} = \frac{1}{2n^c}.$$

By a standard stochastic dominance argument, we get:

$$\Pr\{Y_J = 0\} = \Pr\{Y_J < 1\} \leq \Pr\{\hat{Y}_J < 1\} < \frac{1}{2n^c}.$$

In other words, some process will write a value to $\underline{decision}$ and finish the execution of LEAN after $24\theta^{-12}(c\ln n + \ln 2 + 1)$ scheduler rounds with probability at least $1 - \frac{1}{2n^c}$.

For other processes that have not decided a value, they need to take at most 6 shared memory steps to either write the decision value to $\underline{decision}$ or read a decision value from $\underline{decision}$. If we apply a similar balls in bins analysis as in the proof of Lemma 6.5.4, we will see that all processes take 6 shared memory steps in $\frac{6n}{\theta}((c+2)\ln n + \ln 12)$ scheduler rounds with probability at least $1 - \frac{1}{2n^c}$. Combining the argument above, we will see that all process terminates in $24\theta^{-12}(c\ln n + \ln 2 + 1) + 6n\theta^{-1}((c+2)\ln n + \ln 12)$ scheduler rounds with probability at least $1 - \frac{1}{n^c}$. □

t-TERMINATION

We now take into account crash failures. With crashes, $24\theta^{-12}(c\ln n + \ln 2 + 1) + 6n\theta^{-1}((c+2)\ln n + \ln 12)$ scheduler rounds no longer necessarily guarantee termination with high probability, due to the possibility of advanced processes crashing before they decide. Here we modify our termination bound to include the impact of crashes:

Lemma 6.5.11. *For any constant $c > 0$, the execution of* LEAN *terminates in $O((c\ln n)\cdot(\theta^{-12} + n\theta^{-1}))$ scheduler rounds with at most t crash failures with probability at least $1 - \frac{1}{n^c}$.*

Proof. Suppose we divide $12(T + t)$ scheduler rounds into blocks each with 12 scheduler rounds, where $T = 2\theta^{-12}(c\ln n + \ln 2 + 1) + \frac{n((c+2)\ln n + \ln 24)}{2\theta}$, the number of blocks required by crash-free termination. We are going to prove that $T + t$ blocks are needed for crash termination. Since we assume that crashes are fixed by the scheduler in advance, at most t of these $T + t$ blocks contain a crash event. Therefore, the T blocks that remain are crash free in the sense that every process that is not crashed at the beginning of one of these blocks remains non-crashed throughout all 12 scheduler rounds. We can directly

apply Lemma 6.5.10 to these crash-free blocks. It follows that the probability that no one terminates in one of these crash free blocks is no greater than $\frac{1}{n^c}$. Since the additive term $12t$ will be dominated by the term $\frac{n\theta^{-1}((c+2)\ln n+\ln 12)}{2\theta}$ in T because we assume that $t < n$, the asymptotic bound will be the same. □

6.5.3 SOLVING GENERAL CONSENSUS WITH A STOCHASTIC SCHEDULER

In this part, we are going to present a consensus algorithm called BINARYTREECONSENSUS that uses binary consensus algorithm LEAN as a subroutine. The BINARYTREECONSENSUS algorithm can be used as the BLACKBOXCONSENSUS needed by FAIRCONSENSUS to provide a fair consensus solution that only depends on the stochastic scheduler assumption.

In the following, each process is assigned a unique ID from 1 to n, and process i runs BINARYTREECONSENSUS with initial value v_i and decides one of these values. To simplify notation, we assume that n is a power of 2. Then we can assign a $\log n$-bit string str_i to the i^{th}, and the decimal value of str_i is equal to $i-1$.

The main idea of this algorithm is as follows. We build a binary tree with n leaves. We label every node in the binary tree from 1 to $2n-1$ in top-to-bottom left-to-right order. Note that the root node has label 1, while the i^{th} leaf node from the left that corresponds to process i has label $n+i-1$. We say that the root of the binary tree is on level 1, and both its children are on level 2, and the grandchildren of the root are on level 3, and so on. All leaf nodes are on level $\log n+1$. We use k as the indices for binary tree node labels, and j as the indices for the levels in the binary tree. We assume a shared array $\underline{val}[k]$ for each node $k \in [1, n-1]$ except for leaf nodes in the tree (note that the labels of leaf nodes begin with n), where all process can write their binary strings assigned. BINARYTREECONSENSUS involves $\log n$ instances of LEAN in total, one on each level $j(=1, 2, \ldots, \log n)$ of the binary tree. We may regard the execution of BINARYTREECONSENSUS as walking from

BINARYTREECONSENSUS(i, v_i)
(for all active process i with initial value v_i)

Initially $\underline{v}[i] \leftarrow \perp$ for all $i \in [1, n]$
$\underline{val}[k] \leftarrow \perp$ for all $k \in [1, n-1]$,
$w_i[j] \leftarrow \perp$ for all $j \in [1, \log n]$
$str_i \leftarrow$the $\log n$-bit binary representation of $i - 1$

$id_i \leftarrow str_i$
write v_i to $\underline{v}[i]$
for $j \leftarrow \log n$ **down to** 1 **do**
  $k \leftarrow$ the label of i's ancestor on level j
  write id_i to $\underline{val}[k]$

$decision \leftarrow 0$
for $j \leftarrow 1$ **to** $\log n$ **do**
  if k is not an ancestor of id_i **then**
    $id_i \leftarrow$ **read** $\underline{val}[k]$
  $\tilde{v}_i \leftarrow$ the j^{th} most significant digit of the id_i
  $w_i[j] \leftarrow$ the decision value of LEAN corresponding to level j and called with initial value $\tilde{v}_i$
  $decision \leftarrow decision + w_i[j] \cdot 2^{\log n - j}$
  if $w_i[j] = 0$ **then**
    $k \leftarrow$ the label of k's left child
  else
    $k \leftarrow$ the label of k's right child

$result \leftarrow$ **read** $\underline{v}[decision + 1]$
return $result$

Figure 6.5: The BINARYTREECONSENSUS algorithm.

the root of the binary tree down to a single leaf node, while an instance of LEAN deciding 0 can be seen as a decision to descend to the left subtree and 1 to the right subtree. Therefore, deciding process i's value is equivalent to walking from the root to the i^{th} leaf.

To begin with the execution of algorithm BINARYTREECONSENSUS, process i writes its initial value in $\underline{v}[i]$ and id_i, a binary ID that it supports through the algorithm (initialed by str_i but can change as the algorithm runs), in $\underline{val}[k]$ for all node k on the path from the root to the ith leaf node (including the root). In order to decide which initial value to propose for the instance of LEAN corresponding to tree node k, process i updates id_i by $\underline{val}[k]$ if id_i is not in the current subtree rooted at node k. Then process i proposes 0 if the left subtree of k contains the leaf with label id_i, and proposes 1 otherwise. As the algorithm runs, a $\log n$-bit binary string will be generated after $\log n$ instances of LEAN. The algorithm then decide $\underline{v}[i]$ if the generated binary string is the same as str_i.

CORRECTNESS PROOF

Here we are going to prove that BINARYTREECONSENSUS solves consensus through the proof of Lemma 6.5.12, 6.5.13 and 6.5.14.

Lemma 6.5.12 (Validity)**.** *Suppose* BINARYTREECONSENSUS *returns value* $result$*. Then there exists a process* w *such that* $result = v_w$*.*

Proof. We assume for contradiction that the decision $result$ is not the initial value of any process. According to the pseudocode of BINARYTREECONSENSUS, we know that $0 \leq decision \leq n-1$ by simple calculation, and therefore $result$ comes from the shared array $\underline{v}[1 \ldots n]$ that stores the initial values of processes. Suppose $result = \underline{v}[i]$ for some $i \in [1, n]$. The only case that $result$ is not the initial value of any process is $\underline{v}[i] = \perp$. We will prove that it is impossible to decide $\perp$.

Note that BINARYTREECONSENSUS has the property that when some process has called the instance of LEAN corresponding to node k on level j, the $(j-1)$-bit prefix of the binary string generated by the previous $j-1$ instances of LEAN is the same as str_u for all process u in the subtree rooted at k. This property is straightforward in a full binary tree where the leaves are associated with process $1, 2, \ldots, n$ in left-to-right manner. Then if the instance at k decides 0 (1), then by the validity requirement of LEAN, there exists a process u_j associated with a leaf in k's left (right) subtree such that the j^{th} bit of str_{u_j} is 0 (1). If we focus on $j = \log n$, we will find that $str_{u_{\log n}}$ has exactly the same $\log n$-bit binary representation as str_i. This yields $i = u_{\log n}$, which contradicts our assumption that i is crashed before it can write in $\underline{val}[k_{\log n}]$. □

Lemma 6.5.13 (Agreement)**.** *Suppose during* BINARYTREECONSENSUS*, some process i_1 decides w_1 while another process i_2 decides w_2. It follows that $w_1 = w_2$.*

Proof. Assume for contradiction that $w_1 \neq w_2$. We rewrite w_1 and w_2 as follows:

$$w_1 = \sum_{j=1}^{\log n} w_1[j] \cdot 2^{\log n - j} + 1 \quad w_2 = \sum_{j=1}^{\log n} w_2[j] \cdot 2^{\log n - j} + 1.$$

By the uniqueness of binary notation, there exists $j' \in [1, \log n]$ such that $w_1[j'] \neq w_2[j']$. This occurs only two different values $w_1[j']$ and $w_2[j']$ are decided by algorithm LEAN in same level j', which contradicts the agreement property of LEAN. □

Now we are going to prove a bound on the scheduler rounds required for BINARYTREECONSENSUS to terminate. We note that the more general t-failure probabilistic termination property follows directly from the same style of argument used below.

Lemma 6.5.14. *For any constant $c' > 0$,* BINARYTREECONSENSUS *terminates in $O((c' \log^2 n) \cdot (\theta^{-12} + n\theta^{-1}))$ scheduler rounds with probability at least $1 - \frac{1}{n^{c'}}$ when run with a stochastic scheduler $\mathcal{S}$ with threshold θ.*

The proof of Lemma 6.5.14 requires the following lemma:

Lemma 6.5.15. BINARYTREECONSENSUS *includes at most* $O(\log n)$ *reads and* $O(\log n)$ *writes on the shared memory by each process, in addition to those required by the execution of* LEAN.

Proof. Process i writes $\underline{v}[i]$ as well as $\underline{val}[k]$ for k on the path from the root to the ith leaf. Then the maximum number of writes per process will be $\log n + 2$. On the other hand, the process reads at most one $val[k]$ for every instance k while there are $\log n$ instances in total. Therefore, the maximum number of reads per process is $\log n + 1$ including a final read of the decision value $\underline{v}[decision + 1]$. □

We now return to the proof of Lemma 6.5.14:

Proof (of Lemma 6.5.14). BINARYTREECONSENSUS contains $\log n$ instances of LEAN. We say that an instance of LEAN succeeds if all correct processes terminate and decide on the same value for that instance. BINARYTREECONSENSUS. According to Lemma 6.5.11, an instance of LEAN fails to terminate within $24\theta^{-12}(c \ln n + \ln 2 + 1) + 12n\theta^{-1}((c + 2) \ln n + \ln 24) + 12t$ scheduler rounds with probability no greater than $1/n^c$ for constant $c > 0$. The probability that some instance fails is no greater than $\log n \cdot \frac{1}{n^c} < \frac{1}{n^{c-1}}$ for $c > 1$. In other words, these $\log n$ instances terminate in $(24\theta^{-12}(c \ln n + \ln 2 + 1) + 6n\theta^{-1}((c + 2) \ln n + \ln 12) + 12t) \log n$ scheduler rounds with probability at least $1 - \frac{1}{n^{c-1}}$.

According to Lemma 6.5.15, On the other hand, BINARYTREECONSENSUS requires at most $2 \log n + 3$ more shared memory steps than LEAN by each process. According to the balls in bins argument in the proof of Lemma 6.5.4, these shared memory steps complete in $c\theta^{-1} \ln n(2 \log n + 3)$ scheduler rounds with probability at least $1 - \frac{1}{n^{c-1}}$ for constant $c > 1$. Taking a union bound, $cn\theta^{-1} \ln n(2 \log n + 3)$ scheduler rounds will be required by all processes with probability at least $1 - \frac{1}{n^{c-2}}$ for constant $c > 2$. Note that this term

will be dominated by the termination time of LEAN. Therefore, for any $c' > 0$, the number of scheduler rounds needed for BINARYTREECONSENSUS to terminate with probability at least $1 - \frac{1}{n^{c'}}$ is bounded by $O((c' \log^2 n) \cdot (\theta^{-12} + n\theta^{-1}))$. □

6.6 Application of Fair Consensus in State Machine Replication

One important application of consensus is to implement a fault-tolerant distributed replicated state machine. The replicated state machine strategy was first introduced by Lamport [62]. It is used for constructing fault-tolerant system services in [95, 96], and Herlihy [46] provides an elegant construction of a state machine based on a black box consensus implementation. Lamport [66] also provides a replicated state machine implementation. State machine replication has a lot of application in the maintenance of distributed databases. The strategy of replicated database generalizes replicated state machine approaches, and increases the availability of data [6]. The scalability and benefit of implementing database state machine in a cluster of data servers has been proved in [88]. Byzantium for Byzantine fault tolerant database replication is shown to introduce little performance overhead [33]. In this section, we discuss the advantages of constructing a replicated state machine using a fair consensus algorithm as a black box. The goal is to help motivate the usefulness of fair consensus in practical settings.

In more detail, the goal of a replicated state machine algorithm is to allow a set of fault prone servers to simulate a single state machine. Each server is associated with one or more clients. These clients pass state machine invocations to their server which then passes back a response. Globally, these responses must be consistent with a system that actually contains a single reliable state machine object. A common general strategy to implement a replicated state machine algorithm is to have each server keep a local copy of the state machine. The servers then run an instance of consensus to agree on each state machine operation, ensuring all servers apply the same order of operations to their local replicas.

One possible problem with this kind of implementation is fairness, as a consensus algorithm does not guarantee that every proposed value will be decided. Note that Herlihy [47] gets around this with a complicated procedure in which processes have to help

FAIRRSM(m, r_i) (for process i in round m with request r_i)

do
  $r^{(m)} \leftarrow$ FAIRCONSENSUS(i, m, r_i)
  perform state transition on its state machine replica according to $r^{(m)}$
  $m \leftarrow m + 1$
until $r_i = r^{(m)}$
$s^{(m)} \leftarrow$ current state of the state machine
tell the sender of r_i to transit its state machine replica to state $s^{(m)}$

Figure 6.6: The algorithm of FAIRRSM.

other processes get their old proposals decided. Moreover, this requires keeping a complicated record of request adoption from every process, and implementing complicated record cleaning logic to prevent the record from growing too big. Here we will briefly examine whether the use of fair consensus principles sidesteps these issues, and allow simple implementations of state machines with strong guarantees about all requests to the machines eventually being processed.

FAIRRSM: A REPLICATED STATE MACHINE ALGORITHM USING FAIRCONSENSUS

Here we briefly describe and discuss FAIRRSM, a replicated state machine algorithm that leverages fair consensus. We note that this algorithm is much simpler than existing solutions that do not assume fair consensus (e.g., [46]).

The FAIRRSM algorithm (shown in Figure 6.6) has each server run FAIRCONSENSUS to agree on the next operation to apply to their local state machine replica. The fairness of FAIRCONSENSUS guarantees that if process i does not crash, then its current pending request r_i will eventually be decided and therefore applied to the state machine. According to Theorem 6.5.6, we can find some $m_0 > 0$ such that once the round number $m > m_0$, the execution of FAIRCONSENSUS enters the fair phase. At this point, request r_i will have

probability of $\frac{c}{\tilde{n}}$ to be decided for a constant $c > 0$ by running FAIRCONSENSUS once proposed, where $\tilde{n}$ is the number of correct processes. As a result, the probability that a proposed request r_i is not processed in any state machine round numbered from m to $m + \Delta$ for some $\Delta > 0$ is $(c/\tilde{n})^{\Delta+1}$, which is exponentially small as Δ grows (for a fixed $\tilde{n}$). The FAIRRSM algorithm, therefore, provides a strong probabilistic guarantee of starvation freedom.

CHAPTER 7

CONCLUSION

This thesis studies the fairness of distributed algorithms. Whereas previous notions of fairness focused more on freedom from starvation, we study a more intuitive and stronger notion of fairness which focuses on equality of opportunity. In more detail, in this thesis, we explored this notion of fairness with respect to multiple distributed algorithm problems and models: maximal independent set algorithms and vertex coloring algorithms in the classical synchronous $\mathcal{CONGEST}$ model, blind rate adaptation algorithms in a wireless communication model, contention resolution algorithms in networks with multi-access channels and collision detection, and consensus algorithms in the asynchronous shared memory model. We argue that fair distributed algorithms are potentially beneficial for the construction of robust and effective distributed systems.

Our study of fairness opens many interesting new problems to explore. First of all, there might be alternative ways of defining the "fairness" of graph algorithms, and the corresponding new and fair algorithms are worth studying. Second, in order to achieve fairness, our fair coloring algorithm relaxed the constraint on the number of colors. Another open problem could be, therefore, to reduce the number of colors used by a fair coloring algorithm for graphs with smaller chromatic numbers, such as planar graphs (which are constant colorable).

Our study of fair communication problems also highlights interesting open questions. For example, the order preservation property in wireless communication is closely related to packet coding, and an order preserving rate adaption strategy could make decoding

much easier. This may be beyond the scope of distributed computation, but is still worth exploring. An obvious next step in our study of fair contention resolution is to tackle the final small gap between lower and upper bounds. We predict that our lower bound is in fact tight, while proving this (somewhat tentative) assertion might require more advanced algorithmic techniques. Another open problem is to tackle contention resolution in this setting with respect to expected time of termination. Not much is known about expected time solutions in this case. One reason for this omission is that even without collision detection, the best expected time solutions are very fast, reaching $O(1)$ expected complexity with as few as $\log n$ channels. This leaves only a small band of parameters for which the addition of collision detection might possibly improve the performance. And, of course, it is also interesting to study the fairness of existing contention resolution algorithms in related multi-channel models.

Last but not least, a fair consensus algorithm can provide many nice properties for high-level applications in addition to just replicated state machines. It remains interesting open work to explore places where the use of a fair consensus subroutine simplifies or improves applications.

BIBLIOGRAPHY

[1] N. Abramson. The ALOHA system: another alternative for computer communications. In *Proceedings of the Fall Joint Computer Conference*, 1970.

[2] D. Alistarh, K. Censor-Hillel, and N. Shavit. Are lock-free concurrent algorithms practically wait-free? In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2014.

[3] K. Appel and W. Haken. Every planar map is four colorable. *Contemporary Mathematics*, pages 1–741, 1989.

[4] J. Aspnes. Fast deterministic consensus in a noisy environment. *J. Algorithms*, 45(1):16–39, 2002.

[5] K. Atanassov. On the 37th and the 38th Smarandache problems. *Notes on Number Theory and Discrete Mathematics*, pages 83–85, 1999.

[6] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, chapter 8, pages 183–185. John Wiley & Sons, 2004.

[7] R. Bar-Yehuda, O. Goldreich, and A. Itai. On the time complexity of broadcast in radio networks: an exponential gap between determinism and randomization. In *Proceedings of the International Symposium on Principles of Distributed Computing*, 1987.

[8] L. Barenboim and M. Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC)*, 2008.

[9] L. Barenboim, M. Elkin, S. Pettie, and J. Schneider. The locality of distributed symmetry breaking. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 2012.

[10] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

[11] S. Biaz and S. Wu. Loss differentiated rate adaptation in wireless networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, 2008.

[12] S. Biaz and S. Wu. Rate adaptation algorithms for IEEE 802.11 networks: a survey and comparison. In *Proceedings of IEEE Symposium on Computers and Communications*, 2008.

[13] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, 1987.

[14] L. Bui, A. Eryilmaz, R. Srikant, and X. Wu. Wireless networks with maximal matching-based scheduling. *IEEE/ACM Transactions on Networking*, 16(4):826–839, 2008.

[15] R. Chandra, S. Karanth, T. Moscibroda, V. Navda, J. Padhye, R. Ramjee, and L. Ravindranath. Dircast: a practical and efficient Wi-Fi multicast system. In *Proceedings of the IEEE International Conference on Network Protocols*, 2009.

[16] S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of ACM Symposium on Principles of distributed computing (PODC)*, 2002.

[17] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[18] A. Cornejo and C. Newport. Prioritized gossip in vehicular networks. In *The ACM SIGACT-SIGOPS International Workshop on Foundations of Mobile Computing (DIALM-POMC)*, 2010.

[19] B. P. Crow, I. Widjaja, J. G. Kim, and P. T. Sakai. IEEE 802.11 wireless local area networks. *IEEE Communication Magazine*, 1997.

[20] S. Daum, S. Gilbert, F. Kuhn, and C. Newport. Efficient symmetry breaking in multi-channel radio networks. In *Proceedings of the International Symposium on Distributed Computing*, 2012.

[21] S. Daum, S. Gilbert, F. Kuhn, and C. Newport. Leader election in shared spectrum networks. In *Proceedings of the International Symposium on Principles of Distributed Computing*, 2012.

[22] N. G. de Bruijn. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 10(3):137–138, 1967.

[23] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[24] D. Dolev and H. R. Strong. Distributed commit with bounded waiting. In *Proceedings of the IEEE Symposium on Reliability in Distributed Software and Database System*, 1982.

[25] M. A. Eisenberg and M. R. McGuire. Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM*, 15(11):999, 1972.

[26] M. Farach-Colton, R. J. Fernandes, and M. A. Mosteiro. Lower bounds for clear transmissions in radio networks. In *Proceedings of the Latin American Symposium on Theoretical Informatics*, 2006.

[27] J. Fineman, C. Newport, M. Sherr, and T. Wang. Fair maximal independent set. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2014.

[28] J. T. Fineman, C. Newport, and T. Wang. Brief announcement: Fair maximal independent sets in trees. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing*, 2013.

[29] J. T. Fineman, C. Newport, and T. Wang. Contention resolution on multiple channels with collision detection. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 175–184, 2016.

[30] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossible of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[31] N. Francez. *Fairness*. Springer-Verlag, 1986.

[32] R. Gallager. A perspective on multiaccess channels. *IEEE Transactions on Information Theory*, 31(2):124–142, 1985.

[33] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *Proceedings of the conference on Computer systems*, pages 107–122, 2011.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[35] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 47–63, 1974.

[36] L. Gasieniec, A. Pelc, and D. Peleg. The wakeup problem in synchronous broadcast systems. *SIAM Journal on Discrete Mathematics*, 14(2):207–222, 2001.

[37] S. Gilbert, C. Newport, and T. Wang. Bounds for blind adaptation. In *Proceedings of the International Conference on Principles of Distributed System*, 2015.

[38] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1987.

[39] A. Greenberg and S. Winograd. A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *Journal of the ACM*, 32(3):589–596, 1985.

[40] A. Gudipati and S. Katti. Strider: automatic rate adaptation and collision handling. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.

[41] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed system: a concise guided tour. *Distributed Systems*, pages 33–47, 2000.

[42] B. Hajek and T. van Loon. Decentralized dynamic control of a multiaccess broadcast channel. *IEEE Transactions on Automatic Control*, 27(3):559–569, 1982.

[43] M. Hańćkowiak, M. Karoński, and A. Panconesi. A faster distributed algorithm for computing maximal matchings deterministically. In *Proceedings of ACM Symposium on Principles of distributed computing (PODC)*, pages 219–228, 1999.

[44] D. G. Harris, E. Morsy, G. Pandurang, P. Robinson, and A. Srinivasan. Efficient computation of balanced structures. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, 2013.

[45] L. Harte. *Introduction to data multicasting*. Althos Publishing, 2008.

[46] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[47] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the ACM Symposium on Principles of distributed computing (PODC)*, pages 276–290, 1988.

[48] J. Hirvonen and J. Suomela. Distributed maximal matching: greedy is optimal. In *Proceedings of the ACM Symposium on Principles of distributed computing (PODC)*, 2012.

[49] G. Holland, N. Vaidya, and P. Bahl. A rate-adaptive MAC protocol for multi-hop wireless networks. In *Proceedings of the international conference on Mobile computing and networking (MobiCom)*, pages 236–251, 2001.

[50] T. Jurdzinski and D. R. Kowalski. Distributed backbone structure for algorithms in the SINR model of wireless networks. In *Proceedings of International Symposium on Distributed Computing (DISC)*, 2012.

[51] T. Jurdziński and G. Stachowiak. Probabilistic algorithms for the wakeup problem in single-hop radio networks. In *Proceedings of the International Symposium on Algorithms and Computation*, pages 535–549, 2002.

[52] A. Kamerman and L. Monteban. WaveLAN II: a high-performance wireless LAN for the unlicensed band. *Bell Labs Technical Journal*, 1997.

[53] M. Kaplan. A sufficient condition for non-ergodicity of a Markov Chain. *IEEE Transactions on Information Theory*, IT-25:470–471, July 1979.

[54] A. Kayem, S. Akl, and P. Martin. An independent set approach to solving the collaborative attack problem. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS)*, 2005.

[55] J. Kim, S. Kim, S. Choi, and D. Qiao. CARA: collision-aware rate adaptation for IEEE 802.11 WLANs. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2006.

[56] J. Komlos and A. Greenberg. An asymptotically nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Transactions on Information Theory*, 31(2):302–306, 1985.

[57] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper. JPaxos: state machine replication based on the Paxos protocol, 2011.

[58] F. Kuhn and T. Moscibroda. Initializing newly deployed ad hoc and sensor network. In *Proceeding of the ACM SIGMOBILE International Conference on Mobile Computing and Networking*, 2004.

[59] F. Kuhn, T. Moscibroda, and R. Wattenhofer. Local computation: lower and upper bounds. *CoRR*, abs/1011.5470, 2010.

[60] M. Lacage, M. Manshaei, and T. Turletti. IEEE 802.11 rate adaptation: a practical approach. In *The ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM)*, 2004.

[61] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, pages 95–114, 1978.

[62] L. Lamport. Time, clocks and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[63] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, 1984.

[64] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[65] L. Lamport. Lower bounds on consensus. Available at: http://research.microsoft.com/en-us/um/people/lamport/pubs/consensus-bounds.pdf, 2000.

[66] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[67] L. Lamport. Generalized consensus and Paxos. TechReport MSR-TR-2005-33, Microsoft Research, 2005.

[68] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[69] L. Lamport, M. Pease, and R. Shostak. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[70] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[71] B. Lampson and H. Sturgis. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center, Palo Alto, California, 1979.

[72] B. W. Lampson. How to build a highly available system using consensus. *Distributed Algorithms*, pages 1–17, 1996.

[73] C. Lenzen and R. Wattenhofer. MIS on trees. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC)*, 2011.

[74] Z. Li, A. Das, A. K. Gupta, and S. Nandi. Full auto rate MAC protocol for wireless ad hoc networks. In *IEEE Proceedings on Communication*, 2005.

[75] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, and B. W. Wade. Notes on distributed databases. Technical report, IBM Research Division, 1979.

[76] N. Linial. Locality in distributed graph algorithms. *SIAM Journal of Computing*, 21(1):193–201, 1992.

[77] N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13:441–454, 1993.

[78] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing*, 15(4):1036–1053, November 1986.

[79] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[80] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the ACM Symposium on Principles of distributed computing (PODC)*, pages 137–151, 1987.

[81] N. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1988.

[82] Y. Métivier, J. M. Robson, N. Saheb-Djahromi, and A. Zemmari. An optimal bit complexity randomized distributed MIS algorithm. *Distributed Computing*, 23(5):331–340, 2011.

[83] C. Newport. Radio network lower bounds made easy. In *Proceedings of the International Symposium on Distributed Computing*, 2014.

[84] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2014.

[85] A. Panconesi and R. Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001.

[86] A. Panconesi and A. Srinivasan. On the complexity of distributed network decomposition. *Journal of Algorithms*, 20(2):356–374, 1996.

[87] Q. Pang, V. Leung, and S. C. Liew. A rate adaptation algorithm for IEEE 802.11 WLANs based on MAC-layer loss differentiation. In *Proceedings of IEEE Broadband Wireless Networking Symposium*, 2005.

[88] F. Pedone and R. Guerraoui. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[89] J. Peery, H. Baladrishnan, and D. Shah. Rateless Spinal codes. In *Proceedings of the ACM Workshop on Hot Topics in Networks*, 2011.

[90] D. Peleg. Distributed computing: a locality-sensitive approach. *Society for Industrial and Applied Mathematics (SIAM)*, 5, 2000.

[91] M. O. Rabin. Randomized byzantine generals. In *Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[92] L. G. Robers. ALOHA packet system with and without slots and capture. *ACM SIGCOMM Computer Communication Review*, 5(2):28–42, 1975.

[93] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, 1978.

[94] F. B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM TOCS*, 2(2):145–154, 1984.

[95] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[96] F. B. Schneider. The state machine approach: a tutorial. In *Proc. Workshop on Fault-tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, pages 18–41, 1990.

[97] J. Schneider and R. Wattenhofer. A log-star distributed maximal independent set algorithm for growth-bounded graphs. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC)*, 2008.

[98] M. Snir. On parallel searching. *SIAM Journal on Computing*, 14(3):688–708, 1985.

[99] E. M. Stein and R. Shakarchi. *Real Analysis, Measure Theory, Integration, & Hibert Spaces*, volume 3, page 91. 1st edition, 2005.

[100] M.-T. Sun, L. Huang, A. Arora, and T.-H. Lai. Reliable MAC layer multicast in IEEE 802.11 wireless networks. In *Proceedings of International Conference on Parallel Processing*, 2002.

[101] A. Wald. On cumulative sums of random variables. *The Annals of Mathematical Statistics*, 15(3):283–296, September 1944.

[102] D. E. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM Journal on Computing*, 15(2):468–477, 1986.

[103] S. Wong, H. Yang, S. Lu, and B. Bharghavan. Robust rate adaption for 802.11 wireless networks. In *Proceedings of the international conference on Mobile computing and networking (MobiCom)*, 2006.

[104] J. Xu. *Graph Theory and Its Applications*. University of Science and Technology of China Press, 2010.

[105] D. Yu, Y. Wang, Q.-S. Hua, and F. C. M. Lau. Distributed $(\Delta + 1)$-coloring in the physical model. *Algorithms for Sensor Systems*, pages 146–160, 2012.