Probabilistic Methods in Distributed Computing

Keren Censor Hillel

Probabilistic Methods in Distributed Computing

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Keren Censor Hillel

Submitted to the Senate of the
Technion - Israel Institute of Technology

Elul 5770 Haifa August 2010

Acknowledgements

The research thesis was done under the supervision of Prof. Hagit Attiya in the Department of Computer Science.

I am grateful to Hagit for being much more than a Ph.D. advisor. Her academic parenting has prepared me, in so many aspects, towards continuing my journey in the world of research. I feel fortunate to have had her as my advisor, and could never thank her enough for her endless patience and wise guidance.

I thank my collaborators, Prof. James Aspnes from Yale University and Prof. Hadas Shachnai from the Technion, for many fruitful conversations. Both are a great pleasure to work with, and taught me much.

Prof. Yehuda Afek from Tel-Aviv University and Prof. Yoram Moses from the Technion provided helpful comments and suggestions on an earlier version of this thesis (and in general), for which I thank them both.

Many friends at the Technion helped in making the last years a time that will not be forgotten. I thank David Hay, Yaniv Carmeli, Hanna Mazzawi, Royi Ronen, Tamar Aizikowitz, Sivan Bercovichi, and Sonya Liberman, for their strong friendship, way beyond the uncountable number of joint coffee breaks.

Last, but not least, I thank my parents, Erga and Yair, and my aunt and uncle, Aya and Zvi, for their endless love and support. And Gil, for being my home. This thesis is dedicated to them.

The generous financial help of the Technion, the Neeman and Jacobs-Qualcomm Foundations, and the Adams Fellowship Program of the Israel Academy of Sciences and Humanities, is gratefully acknowledged.

Contents

List of Figures

List of Tables

Abstract

An inherent characteristic of distributed systems is the lack of centralized control, which requires the components to coordinate their actions. This need is abstracted as the *consensus* problem, in which each process has a binary input and should produce a binary output, such that all outputs agree. A difficulty in obtaining consensus arises from the possibility of process failures in practical systems. When combined with the lack of timing assumptions in asynchronous systems, it renders consensus impossible to solve, as proven by Fischer, Lynch, and Paterson in their fundamental impossibility result, which shows that no deterministic algorithm can achieve consensus in an asynchronous system, even if only a single process may fail.

Being a cornerstone in distributed computing, much research has been invested in overcoming this impossibility result. One successful approach is to incorporate randomization into the computation, allowing the processes to terminate with probability 1 instead of in every execution, while never violating agreement.

Randomized consensus is the main subject of this thesis, which investigates algorithms and lower bounds for this problem. In addition, it addresses problems that arise from the study of randomized consensus, including set agreement, and efficient concurrent data structures.

Our main contribution is in settling the total step complexity of randomized consensus, improving both known lower and upper bounds to a tight $\Theta(n^2)$. The upper bound is obtained by presenting a new shared coin algorithm and analyzing its agreement parameter and total step complexity by viewing it as a stochastic process. The lower bound relies on considering a restricted round-based set of executions called layers, and using randomized valency arguments to prevent the algorithm for terminating too quickly. It is shown how to remain with high probability in bivalent configurations, or in null-valent configurations. The latter case is modeled as a one-round coin-flipping game, which is analyzed using an isoperimetric inequality.

The above result closes the question of the total step complexity of randomized consensus under a strong adversary, which can observe the values of all shared variables and the local states of all processes (including results of local coin-flips) before making the next scheduling decision. An additional result we present is a bound on the total number of steps as a function of the probability of termination for randomized consensus under a weak adversary, which must decide upon the entire schedule in advance.

Another complexity measure we investigate is the individual step complexity of any single process. In traditional shared coins a single process may have had to perform all the work by itself, which motivated the design of shared coins that reduce the individual step complexity. This had the price of increasing the total step complexity. In this thesis we show how to combine shared-coin algorithms to enjoy the best of their complexity measures, improving some of the known results.

For the specific model of shared multi-writer multi-reader registers, the question of individual step complexity of randomized consensus has been later settled by constructing a sub-linear approximate counter. This raises the interest in additional sub-linear data structures, and specifically in data structures providing exact values. We present an exact polylogarithmic counter, using a data structure which we call a *max register*, which we implement in a polylogarithmic number of steps per operation. We then construct a framework that uses the polylogarithmic exact counter to obtain a shared-coin algorithm with an optimal individual step complexity of $O(n)$.

Finally, another way to circumvent the impossibility proof of consensus is to allow more choices. This is modeled as the problem of *set agreement*, where the inputs are drawn from a set of size larger than two, and more than one output is allowed. We present randomized algorithms for different parameters of the set agreement problem, which are resilient to any number of failures.

Abbreviations and Notations

n	The number of processes in the system 21
f	The number of possible failures in the system 21
$k+1$	The number of possible inputs to a set agreement algorithm 22
ℓ	The number of allowed outputs to a set agreement algorithm 22
p_i	A process 22
δ	The agreement parameter of a shared-coin algorithm 27
C	A configuration 35
σ	A schedule 35
g	A gate in a circuit 53
L	A layer 94
α	An execution 96
q_k	The probability of not terminating in $k(n-f)$ rounds 100
P_i	A set of processes 100

Chapter 1

Introduction

Coordinating the actions of processes is crucial for virtually all distributed applications, due to the lack of a centralized control. The most basic abstraction for such coordination is the problem of reaching *consensus* on a single binary input given to each process. Consensus is a fundamental task in asynchronous systems, and can be employed to implement arbitrary concurrent objects [47]; consensus is also a key component of the state-machine approach for replicating services [56, 70].

An obstacle in obtaining coordination in distributed systems is the possibility of crash failures, where a crashed process stops taking steps. Applications are required to be resilient to crashes, guaranteeing that non-faulty processes behave correctly. This requirement should hold regardless of the number of failures, i.e., the algorithms should be *wait-free*.

Formally, the consensus problem requires every non-faulty process to eventually output a single binary output (the *termination* condition), such that all outputs are equal (the *agreement* condition). To avoid trivial solutions, this common output must be the input of some process (the *validity* condition).

The quality of algorithms for distributed systems subject to crash failures, and sometimes their existence, is inherently influenced by the timing assumptions provided by the system. Two common types of distributed systems are *synchronous* and *asynchronous* systems. In a synchronous system, the computation proceeds in *rounds*, each round consisting of a single *step* by each non-faulty process, whose type depends on the communication model. In an asynchronous system, no timing assumptions are provided; there are no bounds on the time between two steps of a process, or between steps of different processes. Asynchrony makes it impossible to tell apart a crashed

process from a very slow one.

Perhaps the most celebrated result in distributed computing, known as the FLP impossibility result, first proved by Fischer, Lynch, and Paterson, shows that no deterministic algorithm can achieve consensus in an asynchronous system, even if only a single process may fail [43, 47, 57].

Due to the importance of the consensus problem, much research was invested in trying to circumvent this impossibility result. One successful approach is to relax the termination condition, and allow randomized algorithms in which a non-faulty process terminates only with probability 1. This does not rule out executions in which a process does not terminate, but guarantees that the probability of such executions is 0. We emphasize that the safety requirements, of agreement and validity, remain the same, hence are required to hold in *every* execution.

Randomized consensus is the main subject of this thesis, which investigates algorithms and lower bounds. In addition, it addresses problems that arise from the study of randomized consensus, including set agreement, and efficient concurrent data structures. The remainder of the introduction is dedicated to describing these contributions.

1.1 The Total Step Complexity of Randomized Consensus Under a Strong Adversary

Our main contribution is in proving that $\Theta(n^2)$ is a tight bound for the *total step complexity* of asynchronous randomized consensus. The total step complexity, or the *total work* of a randomized distributed algorithm is the expected total number of steps taken by all the process. Our result is two-fold[1], improving upon both the previously known upper bound of $O(n^2 \log n)$ due to Bracha and Rachman [28] and the previously known lower bound of $\Omega(n^2/\log^2 n)$ due to Aspnes [5]. The communication model addressed is a shared memory system, where processes communicate by reading and writing to *multi-writer multi-reader* registers. The adversary controlling the schedule is a *strong* adversary, which observes all values of shared registers and all local states of processes, including results of local coin-flips, before scheduling the next process to take a step.

Our algorithm relies on a *shared-coin* algorithm [12], as do virtually all randomized consensus algorithms. In a shared-coin algorithm, each process outputs a value -1 or $+1$, and for every

[1] Both results appeared in [15].

$v \in \{-1, +1\}$, there is a probability of at least δ that all processes output the same value v; δ is the *agreement parameter* of the algorithm. Notice that there are no inputs to this procedure. Aspnes and Herlihy [12] have shown that a shared-coin algorithm with agreement parameter δ and expected total work T yields a randomized consensus algorithm with expected total work $O(n^2 + T/\delta)$.

We present a shared-coin algorithm with a *constant* agreement parameter, which leverages a single binary multi-writer register (in addition to n single-writer multi-reader registers). It allows to optimize the trade-off between the total step complexity and the agreement parameter. By viewing any schedule of the algorithm as a stochastic process, and applying Kolmogorov's inequality, we prove that for each possible decision value, all processes output that same value for the shared coin with constant probability. The shared coin has an expected total step complexity of $O(n^2)$. We use this in the Aspnes-Herlihy framework to obtain a randomized consensus algorithm with the same total step complexity of $O(n^2)$.

The matching lower bound is obtained by considering *layered executions*. We focus on configurations at the end of each layer and classify them according to their *valence* [43, 60], namely, the decisions that can be reached in layered extensions. Similarly to notions for deterministic algorithms, a configuration is *univalent* if there is only one possible decision value from all of the extensions of the execution from that configuration. If both decision values are possible then the configuration is *bivalent*. When a decision is reached, the configuration must be *univalent*, so the proof aims to avoid univalent configurations. As opposed to deterministic algorithms, where the valence of a configuration binds the extension to reach a certain decision value v, in a randomized algorithm the valence only implies that some execution will decide v with high probability [5]. This leaves the possibility of *null-valent* configurations, from which no decision value is reached with high probability. When the configuration is null-valent, we derive an isoperimetric inequality in order to control a one-round coin-flipping game for reaching another null-valent configuration.

Our general proof structure follows a proof by [23] of an $\Omega(\sqrt{n/\log n})$ lower bound on the expected number of rounds in a randomized consensus algorithm for the *synchronous message passing* model. In particular, like them, we treat null-valent configurations by considering one-round coin-flipping games and applying an isoperimetric inequality. Unlike their proof, our proof handles the more complicated shared-memory model and exploits the fact that in an asynchronous

system, processes can be hidden in a one-round coin flipping game without having to fail for the rest of the execution.[2]

The layered approach was introduced by [60], who employed it to study deterministic consensus. They showed that the layered approach can unify the impossibility proof for asynchronous consensus [43, 47, 57] with the lower bound on the number of rounds needed for solving synchronous consensus [37, 42]. Their work considered the message-passing model as well as the shared-memory model with *single-writer* registers. We take the layered approach one step further and extend it to randomized algorithms, deriving the lower bound on their total step complexity within the same framework as the results for deterministic algorithms. Besides incorporating randomization into the layered model, our proof also deals with the challenge of allowing processes to access *multi-writer* registers.

1.2 A Lower Bound for Randomized Consensus Under a Weak Adversary

The previous lower bound, accompanied by our algorithm, closes the question of the total step complexity of randomized consensus under a strong adversary. However, there are additional types of adversaries when randomized algorithms are considered. We provide a second lower bound for randomized consensus algorithms, under the control of a *weak adversary* that chooses the entire schedule in advance, without observing any progress of the execution.

This is not a lower bound on the total step complexity. Instead, we make use of the observation that in typical randomized algorithms for consensus, the probability of *not* terminating in agreement decreases as the execution progresses, becoming smaller as processes perform more steps. Our work shows that this behavior is inherent, by proving lower bounds on the probability of termination when the step complexity is bounded.

We prove that for every integer k, the probability that an f-resilient randomized consensus algorithm of n processes does not terminate after $k(n-f)$ steps is at least $\frac{1}{c^k}$, where c is a constant if

[2]Hiding processes in a layer can be described as a round-based model with *mobile* failures, where a process that fails in a certain round may still take steps in further rounds [68]. The equivalence between this model and the asynchronous model is discussed by [64].

$\lceil \frac{n}{f} \rceil$ is a constant[3]. The result holds for asynchronous message-passing systems and asynchronous shared-memory systems (using reads and writes), albeit with different constants. While the same general proof structure applies in both cases, it is accomplished differently in the message-passing and the shared-memory models; the latter case is further complicated due to the adversary's weakness.

For the message-passing model, our proof extends and improves on a result of Chor, Merritt and Shmoys [36] for *synchronous* message-passing systems. They show that the probability that a randomized consensus algorithm does not terminate after k rounds (and $k(n-f)$ steps) is at least $\frac{1}{c \cdot k^k}$. (A similar result is attributed to Karlin and Yao [53].) The proof rests on considering a specific chain of *indistinguishable* executions and showing a correlation between the termination probability and the length of this chain (the number of executions in it), which in turn depends on the number of rounds. The chain is taken from the proof of the rounds lower bound for (deterministic) consensus [37, 42] (cf. [18, Chapter 5]); since the chain is determined in advance, i.e., regardless of the algorithm's transitions, the lower bound is derived with a weak adversary.

Our first contribution in this context, for the message-passing model, improves on this lower bound by exploiting the fact that asynchrony allows to construct "shorter" indistinguishability chains.

The lower bound can be extended to Monte-Carlo algorithms that always terminate, at the cost of compromising the agreement property. If an asynchronous message-passing algorithm always terminates within $k(n-f)$ steps, then the probability for disagreement is at least $\frac{1}{c^k}$, where c is a constant if $\lceil \frac{n}{f} \rceil$ is a constant. This lower bound can be compared to the recent consensus algorithms of Kapron et al. [52] for the message-passing model. One algorithm always terminates within $\mathrm{polylog}(n)$ asynchronous rounds, and has a probability $\frac{1}{\mathrm{polylog}(n)}$ for disagreeing, while the other terminates within $2^{\Theta(\log^7 n)}$ asynchronous rounds, and has a probability $\frac{1}{\mathrm{poly}(n)}$ for disagreeing.

A common theme in both of our lower bounds is the adaptation of the layering technique to randomized algorithms, and the manipulation of layers in order to argue about indistinguishable executions. This integrates the layering approach, which helps in obtaining simple bounds when the model is asynchronous, with the randomization used by the algorithms.

In principle, the lower bound scheme can be extended to the shared-memory model by focusing

[3]This result appeared in [14].

on such layered executions. However, our severely handicapped adversarial model poses a couple of technical challenges. First, while in the message-passing model each step can be assumed to send messages to all processes, in a shared-memory event, a process chooses which register to access and whether to read from it or write to it. A very weak adversary, as we use for our lower bounds, must find a way to make its scheduling decisions in advance without even knowing what type of step the process will take. Second, the proof scheme requires schedules to be determined independently of the coin-flips. The latter difficulty cannot be alleviated even by assuming an adaptive adversary that may schedule the processes according to the execution so far, since future coin flips may lead to different types of steps.

We manage to extend the lower bound scheme to the shared-memory model, by first simplifying the model, assuming that processes either write to *single-writer* registers or perform a *cheap snapshot* operation, reading all the registers at once. By further assuming that an algorithm regularly alternates between writes and cheap snapshots, we make processes' steps predictable, allowing a weak adversary to construct indistinguishability chains. The lower bound is extended to hold for multi-writer registers by reduction; while ordinary simulations of multi-writer registers using single-writer registers have $O(n)$ overhead (which would significantly deteriorate the lower bound), cheap snapshots admit a simulation with constant overhead.

1.3 Combining Shared Coins

In addition to investigating the total number of steps of randomized consensus algorithms, we address their *individual step complexity*. The individual step complexity, or the *individual work* of a randomized distributed algorithm is the expected number of steps taken by any single process. Having a small individual step complexity is not an immediate consequence of having a small total step complexity, because a process running alone may have to perform all the work by itself.

As is the case for the total step complexity, the framework of Aspnes and Herlihy [12] shows that a shared-coin algorithm with agreement parameter δ and expected total work I yields a randomized consensus algorithm with expected total work $O(n + I/\delta)$. Previously, the best known individual work was achieved by the algorithm of Aspnes and Waarts [13], who presented a shared coin algorithm in which each process performs $O(n \log^2 n)$ expected number of steps, which is

significantly lower than the total step complexity bounds. However, by simply multiplying the individual work by n, their algorithm increases the total work to $O(n^2 \log^2 n)$. This led Aspnes and Waarts to ask whether this tradeoff is inherent.

We show that there is no such tradeoff, and in fact, each complexity measure can be optimized separately. This result is achieved by showing that shared-coin algorithms can be *interleaved* in a way that obtains the best of their complexity figures, without harming the agreement parameter.[4]

Given the power of the adversary to observe both algorithms and adjust their scheduling, it is not obvious that this is the case, and indeed, following [71], the work of Lynch, Segala, and Vaandrager [59] shows that undesired effects may follow from the interaction of an adaptive adversary and composed probabilistic algorithms. Nonetheless, we can show that in the particular case of shared-coin algorithms, two algorithms with agreement parameter δ_A and δ_B can be composed with sufficient independence such that the combined algorithm terminates with the minimum of each of the algorithms' complexities (e.g., in *both* individual and total work), while obtaining an agreement parameter of $\delta_A \cdot \delta_B$.

An immediate application of our result shows that the shared coin algorithm of Bracha and Rachman can be interleaved with the algorithm of Aspnes and Waarts, to get an algorithm with both $O(n \log^2 n)$ individual work and $O(n^2 \log n)$ total work. This implies that wait-free randomized consensus can be solved with $O(n \log^2 n)$ expected individual work and $O(n^2 \log n)$ expected total work, using *single-writer* multi-reader registers. These are currently the best complexities known for this model.

Our result has other applications for combining shared coins in order to enjoy the best of more than one complexity measure. For example, Saks, Shavit, and Woll [66] presented three shared-coin algorithms, each having a good complexity for a different scenario. The complexity measure they consider is of *time units*: one time unit is defined to be a minimal interval in the execution of the algorithm during which each non-faulty processor executes at least one step. The first is a wait-free algorithm which takes $O(\frac{n^3}{n-f})$ time, where f is the actual number of faulty processes. In the second algorithm, the time is $O(\log n)$ in executions with at most $f = \sqrt{n}$ faulty processes, and in the third algorithm the time is $O(1)$ in failure-free executions. All three shared coin algorithms have constant agreement parameters, and Saks, Shavit, and Woll claim that they can be interleaved to

[4]This result appeared in [10].

yield one algorithm with a constant agreement parameter that enjoys all of the above complexities. Our argument is the first to prove this claim.

Our result holds also for the shared-memory model with *multi-writer* multi-reader registers. In this model, building upon our $O(n^2)$ shared coin and the weighted voting scheme of [13], it has been shown [8] that the individual work can be further reduced to $O(n \log n)$; again, this came at a cost of increasing the total work to $O(n^2 \log n)$, which can be avoided by combining the $O(n \log n)$ individual work shared coin with our $O(n^2)$ total work shared coin. However, this was superseded by Aspnes and Censor [11], who gave a shared-coin algorithm with $O(n)$ individual work and (immediately) $O(n^2)$ total work, which is optimal due to our $O(n^2)$ lower bound on total work. This shared coin is established based on an approximate *sub-linear counter*, allowing to reduce the individual work to $O(n)$.

1.4 A Polylogarithmic Counter for an Optimal Individual Step Complexity

While an approximate sub-linear counter suffices for closing the question of the individual step complexity of randomized consensus algorithms, this leads us to ask whether there exist better counters, and specifically an *exact* sub-linear counter. Exploring this question resulted in an affirmative answer, as well as constructions of additional polylogarithmic concurrent data structures.

One successful approach to building concurrent data structures is to employ the *atomic snapshot* abstraction [2]. An atomic snapshot object is composed of components, each of which typically is updated by a different processes; the components can be atomically scanned. By applying a specific function to the scanned components, we can provide a specific data structure. For example, to obtain a *max register*, supporting a write operation and a `ReadMax` operation that returns the largest value previously written, the function returns the component with the maximum value; to obtain a *counter*, supporting an increment operation and a `ReadCounter` operation, the function adds up the contribution from each process.

Constructions of exact counters take a linear (in n) number of steps. This is due to the cost of implementing atomic snapshots [49]. Indeed, Jayanti, Tan, and Toueg [51] show that operations must take $\Omega(n)$ space and $\Omega(n)$ steps in the worst case, for many common data structures, including

max registers and counters. This seems to indicate that we cannot do better than snapshots for exact counting.

However, careful inspection of Jayanti, Tan, and Toueg's lower bound proof reveals that it holds only when there are numerous operations on the data structure. Thus, it does not rule out the possibility of having sub-linear algorithms when the number of operations is bounded, or, more generally, the existence of algorithms whose complexity depends on the number of operations. Such data structures are useful for many applications, either because they have a limited life time, or because several instances of the data structure can be used.

We present polylogarithmic implementations of key data structures with a bounded number of values m[5]. The cornerstone of our constructions, and our first example, is an implementation of a max register that beats the $\Omega(n)$ lower bound of [51] when $\log m$ is $o(n)$. If the number of values is bounded by m, its cost per operation is $O(\log m)$; for an unbounded set of values, the cost is $O(\min(\log v, n))$, where v is the value of the register.

Instead of simply summing the individual process contributions, as in a snapshot-based implementation of a counter, we can use a tree of max registers to compute this sum: take an $O(\log n)$ depth tree of two-input adders, where the output of each adder is a max register. To increment, walk up the tree recomputing all values on the path. The cost of a read operation is $O(\min(\log v, n))$, where v is the current value of counter, and the cost of an increment operation is $O(\min(\log n \log v, n))$. When the number of increments is polynomial, this has $O(\log^2 n)$ cost, which is an exponential improvement from the trivial upper bound of $O(n)$ using snapshots. The resulting counter is wait-free and linearizable.

More generally, we show how a max register can be used to transform any monotone circuit into a wait-free concurrent data structure that provides write operations setting the inputs to the circuit and a read operation that returns the value of the circuit on the largest input values previously supplied. Monotone circuits expose the parallelism inherent in the dependency of the data structure's values on the arguments to the operations. Formally, a *monotone circuit* computes a function over some finite alphabet of size m, which is assumed to be totally ordered. The circuit is represented by a directed acyclic graph where each node corresponds to a gate that computes a function of the outputs of its predecessors. Nodes with in-degree zero are input nodes; nodes with

[5]These results appeared in [9].

out-degree zero are output nodes. Each gate g, with k inputs, computes some monotone function f_g of its inputs. Monotonicity means that if $x_i \geq y_i$ for all i, then $f_g(x_1, \ldots, x_k) \geq f_g(y_1, \ldots, y_k)$.

The cost of writing a new value to an input to the circuit is bounded by $O(Sd \min(\lceil \lg m \rceil, n)$, where m is the size of the alphabet for the circuit, d is the number of inputs to each gate, and S is the number of gates whose value changes as the result of the write. The cost of reading the output value is $\min(\lceil \lg m \rceil, O(n))$. While the resulting data structure is not linearizable in general, it satisfies a weaker but natural consistency condition, called *monotone consistency*, which we show is still useful for many applications.

1.5 Randomized Consensus with Optimal Individual Work

We show how to use a polylogarithmic exact counter for obtaining randomized consensus with an optimal individual step complexity of $O(n)$, by constructing a shared coin with that individual step complexity, and using the Aspnes-Herlihy framework discussed earlier.[6]

Essentially all known shared coins are based on random voting, with some variation in how votes are collected and how termination is detected.

Our shared coin is based on the *weighted voting* approach pioneered in the $O(n \log^2 n)$ individual-work algorithm of Aspnes and Waarts [13], where a process that has already cast many votes becomes impatient and starts casting votes with higher weight. We combined this with the termination bit of the shared coin of $O(n^2)$ total work, to allow a detection of termination to be spread fast among the processes.

Still, detecting for the first time that the threshold of number of votes has been reached is a counting problem: using a standard counter with $O(n)$-operation reads means that the threshold can be checked only occasionally without blowing up the cost. By applying our sub-linear counter, a process can carry out as many as $\Theta(\log n)$ counter reads within the $O(n)$ time bound. This gives an $O(n)$ individual step complexity for the shared coin algorithm, and thus $O(n)$ individual step complexity for randomized consensus.

[6] This is an adaptation of the result that appeared in [11].

1.6 Randomized Set Agreement

The last contribution of this thesis is for the problem of *set agreement*. In the set-agreement problem the processes start with input values in $\{0, \ldots, k\}$, instead of binary inputs, and should produce output values such that there are at most ℓ different outputs, for some $\ell < k$. As in the case of consensus, the termination condition requires every non-faulty process to eventually decide, and to avoid trivial solutions, the validity condition requires each output value to be the input value of some process.

The problem of set agreement was introduced by Chaudhuri [32] as a generalization of the consensus problem, in order to deterministically overcome the well-known FLP impossibility of solving consensus deterministically in an asynchronous system which allows even one crash-failure. Chaudhuri showed that if the bound f on the number of faulty processes is smaller than ℓ, then set agreement can be solved by a deterministic algorithm. Later, it was shown by Borowsky and Gafni [27], Herlihy and Shavit [48], and Saks and Zaharoglou [67], using topological arguments, that set agreement cannot be solved deterministically in an asynchronous system if $f \geq \ell$, and in particular it does not have a wait-free solution.

As in the case of consensus, randomization also allows to overcome the impossibility result for set agreement. We present randomized wait-free algorithms for solving set agreement in an asynchronous shared-memory system.[7] First, we generalize the definition of a shared-coin algorithm, and define *multi-sided shared-coin* algorithms. In such an algorithm, each process outputs one of $k + 1$ values (instead of one of two values as in a regular shared-coin), such that each subset of k values has probability at least δ for containing the outputs of all the processes. In other words, each value has probability at least δ of *not* being the output of any process. We then extend the Aspnes-Herlihy framework for using a shared coin for obtaining a randomized consensus algorithm [12], and show how to use any multi-sided shared coin in order to obtain a randomized set-agreement algorithm, for agreeing on k values out of $k + 1$.

Next, we present an implementation of a $(k + 1)$-sided shared-coin algorithm which has a constant agreement parameter, $O(n^2/k)$ total step complexity, and $O(n/k)$ individual step complexity. We then derive a set-agreement algorithm from the $(k + 1)$-sided shared coin using the above framework.

[7] As appeared in [30].

In addition, we present set-agreement algorithms that are designed for agreeing on ℓ values out of $k+1$, for $\ell < k$. In particular, they can be used for the case $\ell = 1$, where the processes agree on the same value, i.e., for *multi-valued consensus*. By definition, solving multi-valued consensus is at least as hard as solving *binary consensus* (where the inputs are in the set $\{0, 1\}$, i.e., $k = 1$), and potentially harder. One algorithm uses multi-sided shared coins, while the other two embed binary consensus algorithms in various ways.

1.7 Overview of the Thesis

Additional background is provided in Chapter 2, followed by a formal description of the model in Chapter 3.

The technical presentation has a different structure than the introduction. It is partitioned into two parts; one studies algorithms and the second is dedicated to lower bounds, in order to emphasize the common techniques.

First, Part I presents the algorithms discussed in the introduction. It begins with Chapter 4, presenting our shared-coin algorithm used to derive randomized consensus with $O(n^2)$ total step complexity. Next, Chapter 5 proves our results regarding the interleaving of shared-coin algorithms. Chapter 6 presents our polylogarithmic concurrent data structures, and Chapter 7 shows how to use a polylogarithmic counter to obtain a shared coin with an optimal individual step complexity of $O(n)$. Finally, we conclude the algorithms part in Chapter 8, with algorithms for set agreement.

The second part of the thesis, Part II, contains the lower bounds for randomized consensus. We begin by laying down the framework of manipulating layered executions in Chapter 9, followed by our lower bound under a weak adversary in Chapter 10, and then our lower bound under a strong adversary in Chapter 11.

We complete the thesis with a discussion in Chapter 12.

Chapter 2

Related Work

In this chapter we survey the background related to our work. We begin with a short historical overview of randomized consensus algorithms for shared memory. An excellent survey of the state of the art prior to 2003 appears in [6].

Following the first randomized consensus algorithm of Ben-Or [24], many randomized consensus algorithms have been suggested, in different communication models and under various assumptions about the adversary. In particular, algorithms were designed to solve randomized consensus in asynchronous shared-memory systems, against a strong adversary that can observe the results of local coin flips before scheduling the processes. Abrahamson [1] presented a randomized algorithm for solving consensus in asynchronous systems using shared memory, whose total work is exponential in n, the number of processes. The first polynomial algorithm for solving randomized consensus under the control of a strong adversary was presented by Aspnes and Herlihy [12]. They described an algorithm that has a total work of $O(n^4)$. The amount of memory required by this algorithm was later bounded by Attiya, Dolev, and Shavit [16]. Aspnes [4] presented an algorithm for randomized consensus with $O(n^4)$ total work, which also uses bounded space. These algorithms were followed by an algorithm of Saks, Shavit and Woll [66] with $O(n^3)$ total work, and an algorithm of Bracha and Rachman [28] with $O(n^2 \log n)$ total work, where the latter was previously the best known total step complexity. A lower bound of $\Omega(\frac{n^2}{\log^2 n})$ on the expected total number of coin flips was proved by Aspnes in [5]; this implies the same lower bound on the total step complexity.

The previous $\Omega(\frac{n^2}{\log^2 n})$ lower bound [5] for randomized consensus under a strong adversary

relies on a lower bound for coin flipping games. In this proof, the adversary schedules processes step-by-step, and the results of the games are analyzed through hyperbolic functions. In contrast, our approach considers only the configurations at the end of layers, allowing powerful results about product probability spaces to be applied, and streamlining the analysis of the behavior of executions.

There is an extensive literature on randomized agreement algorithms for message passing systems under a weak adversary. Recent papers in this area provide algorithms for agreement in the presence of Byzantine processes in *full information* models, where the adversary is computationally unbounded. See [25, 44, 54] for a more detailed description and references.

To the best of our knowledge, there are no other lower bounds on randomized consensus in shared-memory systems under a weak adversary. There are several algorithms assuming a *value-oblivious* adversary, which may determine the schedule adaptively based on the functional description of past and pending operations, but cannot observe any value of any register nor results of local coin-flips. This model is clearly stronger than the adversary we employ, and hence our lower bounds apply to it as well. The algorithms differ by the type of shared registers they use [20–22, 31]. For single-writer multi-reader registers, Aumann and Bender [21] give a consensus algorithm that has probability of at most $\frac{1}{n^c}$ of not terminating within $O(n \log^2 n)$ steps. For multi-writer multi-reader registers, Aumann [20] shows a consensus algorithm in which the probability of not terminating in k iterations (and $O(k \cdot n)$ steps) is at most $(3/4)^k$.

Chandra [31] gives an algorithm with $O(\log^2 n)$ *individual* step complexity, assuming an intermediate adversary that cannot see the outcome of a coin-flip until it is read by some process. Aumann and Kapah-Levy [22] give an algorithm with $O(n \log n \exp(2\sqrt{\mathrm{polylog} n}))$ total step complexity, using single-writer single-reader registers, and assuming a value-oblivious adversary.

An algorithm with $O(n \log \log n)$ total step complexity against a weak adversary was given by Cheung [34], which considers a model with a stronger assumption that a write operation occurs atomically after a local coin-flip. It improves upon earlier work by Chor, Israeli, and Li [35], who provide an algorithm with $O(n^2)$ total step complexity using a slightly different atomicity assumption.

We now turn our attention to randomized set agreement. Previous randomized agreement algorithms for asynchronous shared-memory systems under a strong adversary are for the specific case

of binary consensus, as discussed earlier, with the optimal individual and total step complexities being $O(n)$ and $O(n^2)$, respectively [11, 15].

Unlike the shared-memory model, several set-agreement algorithms for asynchronous *message-passing* systems have been proposed. Mostefaoui et al. [62] use binary consensus to construct a multi-valued consensus algorithm for message-passing systems. This work assumes reliable broadcast. In the same model, Zhang and Chen [74] present improved algorithms, which reduce the number of binary consensus instances that are required. Under the above assumption, Ezhilchelvan et al. [39] also present a randomized multi-valued consensus algorithm, while Mostefaoui and Raynal [61] present a randomized set-agreement algorithm for agreeing on ℓ values out of n. The above algorithms require a bound on the number of failures $f < n/2$, a restriction that can be avoided in the shared-memory model. Moreover, there is an exponentially small agreement parameter for the shared coins that are used, which causes the expected number of phases until agreement is reached to be large.

There is additional literature on set agreement in *synchronous* systems. An important result is by Chaudhuri et al. [33] who show that $f/k + 1$ is the number of rounds needed for solving k-set agreement in shared memory. Raynal and Travers [65] propose new algorithms in addition to a good survey on synchronous set-agreement algorithms.

Chapter 3

The Basic Model and Definitions

This thesis considers a standard model of an asynchronous shared-memory system, where n processes, $p_1, \ldots, p_n$, communicate by reading and writing to shared multi-writer multi-reader (atomic) registers (cf. [19, 58]). In some cases, which will be explicitly noted, it will be more convenient to consider the set of processes as $\{p_0, \ldots, p_{n-1}\}$.

Each *step* consists of some local computation, including an arbitrary number of local coin flips (possibly biased) and one shared memory *event*, which is a read or a write to some register. Processes may fail by crashing, in which case they do not take any further steps.

The system is *asynchronous*, meaning that the steps of processes are scheduled according to an adversary. This implies that there are no timing assumptions, and specifically no bounds on the time between two steps of a process, or between steps of different processes.

An algorithm is *f-tolerant* if it satisfies the requirements of a problem in a system where at most f processes can fail by crashing. In some cases we will require our algorithms to be *wait-free*, i.e., to be correct even if $f = n - 1$ processes may fail during an execution.

Since our algorithms are randomized, different assumptions on the power of the adversary may yield different results. Throughout most of this thesis, we assume that the interleaving of processes' events is governed by a *strong* adversary that observes the results of the local coin flips before scheduling the next event; in particular, it may observe a coin-flip and, based on its outcome, choose whether or not that process may proceed with its next shared-memory operation.

The only exception to this is in the lower bound presented in Chapter 10, where we assume a weaker adversary. It is explicitly defined before being used.

The complexity measures we consider are the following. The *total step complexity* or *total work* of an algorithm is the expected total number of steps taken by all the processes during a worst-case execution of the algorithm. Similarly, the *individual step complexity* or *individual work* of an algorithm is the expected number of steps taken by any single process during a worst-case execution of the algorithm.

Following are the formal definitions of the main problems addressed.

Definition 3.1 *In a consensus algorithm, each process p_i has an input value x_i, and should decide on an output value y_i. An algorithm for solving randomized consensus satisfies the following requirements.*

Agreement: *For every two non-faulty processes p_i and p_j, if y_i and y_j are assigned then $y_i = y_j$.*
Validity: *For every non-faulty process p_i, if y_i is assigned then $y_i = x_j$ for some process p_j.*
Termination: *With probability 1, every non-faulty process p_i eventually assigns a value to y_i.*

Note that the safety requirements of agreement and validity need to always hold, while termination only needs to hold with probability 1. The problem of set agreement is defined similarly, as follows.

Definition 3.2 *In an algorithm for solving $(\ell, k+1, n)$*-agreement *each process p_i has an input value x_i in $\{0, \ldots, k\}$ and should decide on an output value y_i such that the following conditions hold:*

Set Agreement: *There are at most ℓ different outputs.*
Validity: *For every non-faulty process p_i, if y_i is assigned then $y_i = x_j$ for some process p_j.*
Termination: *With probability 1, every non-faulty process p_i eventually assigns a value to y_i.*

We sometimes use the term *set agreement* without parameters for abbreviation. The particular case in which $\ell = 1, k > 1$ is the problem of *multi-valued consensus*, while in case $\ell = k = 1$ we have *binary consensus*.

For completeness, we redefine the notion of a *shared-coin* algorithm, as discussed in Chapter 1. In a shared-coin algorithm the processes do not have inputs and each process should output a value -1 or $+1$. The *agreement parameter* of a shared-coin algorithm is the maximal value δ such that for every $v \in \{-1, +1\}$, there is a probability of at least δ that all processes output the same value v.

Algorithm 3.1 A randomized consensus algorithm from a shared coin, code for p_i

Local variables: $r = 1$, $decide =$ false, $myValue = input$,
$myPropose = [\,]$, $myCheck = [\,]$
Shared arrays: $Propose[\,][0..1]$, $Check[\,]$[agree, disagree]

```
1:  while decide == false
2:      Propose[r][myValue] = true
3:      myPropose = collect(Propose[r])
4:      if there is only one value in myPropose
5:          Check[r][agree] = ⟨true, myValue⟩
6:      else
7:          Check[r][disagree] = true
8:      myCheck = collect(Check[r])
9:      if myCheck[disagree] == false
10:         decide = true
11:     else if myCheck[agree] == ⟨true, v⟩
12:         myValue = v
13:     else if myCheck[agree] == false
14:         myValue = sharedCoin[r]
15:     r = r + 1
16: end while
17: return myValue
```

Algorithm 3.1 gives the framework of Aspnes and Herlihy [12] for deriving a randomized binary consensus algorithm from a shared coin called sharedCoin, with an agreement parameter δ. It follows the presentation given by Saks, Shavit, and Woll [66]. However, the complexity is improved by using multi-writer registers, based on the construction of Cheung [34].

The basic idea is that agreement is easy to detect, while the main challenge is in obtaining it. The randomized consensus algorithm proceeds by (asynchronous) phases, in which the processes try to obtain agreement (sometimes using the shared coin) and then detect whether agreement has been reached. Each process p writes its own preference to a shared array $Propose$, checks if the preferences agree on one value, and notes this in another shared array $Check$. If p indeed sees

agreement, it also notes its preference in $Check$.

Process p then checks the agreement array $Check$. If p does not observe a note of disagreement, it decides on the value of its preference. Otherwise, if there is a note of disagreement, but also a note of agreement, p adopts the value associated with the agreement notification as preference for the next phase. Finally, if there is only a notification of disagreement, the process participates in a shared-coin algorithm and prefers the output of the shared coin.

In every phase, it is guaranteed that only one value can be decided upon, and if some process decides then all others either decide as well or adopt this decision value for the next phase. Also, only one value can be adopted by processes for the next phase. This implies that for the processes to have different preferences for the next phase, it must be that at least one process executes the shared-coin algorithm. But with probability at least δ all the processes that run the shared-coin algorithm output the same value. This is true even if other processes have adopted a value without participating in the shared-coin algorithm, since for each of the two values we have a probability δ for agreeing on that value. Therefore, after the first phase, in which the preferences are given by the adversary, the number of phases until agreement is reached is a geometric random variable, and so its expectation is $1/\delta$.

For a complete proof, see Chapter 8, Section 8.1, where a generalized framework is presented, allowing to obtain a randomized set-agreement algorithm from a *multi-sided shared coin*.

Part I

Algorithms

Chapter 4

A Randomized Consensus Algorithm

As discussed in Chapter 1, Aspnes and Herlihy [12] have shown that a shared-coin algorithm with agreement parameter δ, expected individual work I, and expected total work T, yields a randomized consensus algorithm with expected individual work $O(n + I/\delta)$ and expected total work $O(n^2 + T/\delta)$. Our randomized consensus algorithm is obtained by constructing a shared coin, as described and proved in the following section. For an explicit reduction from a randomized consensus algorithm to a shared coin algorithm, we refer the reader to Section 8.1, where a generalized and slightly improved framework is presented.

4.1 A Shared Coin with $O(n^2)$ Total Work

This section presents a randomized consensus algorithm with $O(n^2)$ total step complexity, by introducing a shared coin algorithm with a *constant* agreement parameter and $O(n^2)$ total step complexity. Using a shared coin algorithm with $O(n^2)$ total step complexity and a constant agreement parameter in the scheme of [12], implies a randomized consensus algorithm with $O(n^2)$ total step complexity.

As in previous shared coin algorithms [28, 66], in our algorithm the processes flip coins until the amount of coins that were flipped reaches a certain threshold. An array of n single-writer multi-reader registers records the number of coins each process has flipped, and their sum. A process reads the whole array in order to track the total number of coins that were flipped.

Each process decides on the value of the majority of the coin flips it reads. Our goal is for the

processes to read similar sets of coins, in order to agree on the same majority value. For this to happen, we bound the total number of coins that are flipped (by any process) after some process observes that the threshold was exceeded. A very simple way to guarantee this property is to have processes frequently read the array in order to detect quickly that the threshold was reached. This, however, increases the total step complexity. Therefore, as in previous shared coin algorithms, we have to resolve the tradeoff between achieving a small total step complexity and a large (constant) agreement parameter.

The novel idea of our algorithm in order to overcome this conflict, is to utilize a multi-writer register called *done* that serves as a binary termination flag; it is initialized to false. A process that detects that enough coins were flipped, sets *done* to true. This allows a process to read the array only once in every n of its local coin flips, but check the register *done* before each local coin flip.

The pseudocode appears in Algorithm 4.1. In addition to the binary register *done*, it uses an array A of n single-writer multi-reader registers, each with the following components (all initialized to 0):

count: how many flips the process performed so far.

sum: the sum of coin flips values so far.

Each process keeps a local copy a of the array A. The collect operation in lines 6 and 8 is an abbreviation for n read operations of the array A.

For the proof, fix an execution α of the algorithm. We will show that all processes that terminate agree on the value 1 for the shared coin with constant probability; by symmetry, the same probability holds for agreeing on -1, which implies that the algorithm has a constant agreement parameter.

The total count of a specific collect is the sum of $a[1].count, \ldots, a[n].count$, as read in this collect. Note that the total count in Line 8 is ignored, but it can still be used for the purpose of the proof.

Although the algorithm only maintains the counts and sums of coin flips, we can (externally) associate them with the set of coin flips they reflect; we denote by F_C the collection of core coin flips that are written in the shared memory by the first time that true is written to *done*. The size of F_C can easily be bounded, since each process flips at most n coins before checking A.

Lemma 4.1 *F_C contains at least n^2 coins and at most $2n^2$ coins.*

Algorithm 4.1 Shared coin algorithm: code for process p_i.

local integer num, initially 0

array a[1..n]

1: while not *done* do

2: $num++$

3: flip = random(-1,$+1$) // a fair local coin

4: A[i].$\langle count, sum\rangle$ = $\langle count++, sum + flip\rangle$ // atomically

5: if $num == 0 \bmod n$ then // check if time to terminate

6: a = collect A // n read operations

7: if $a[1].count + ... + a[n].count \geq n^2$ then *done* = true // raise termination flag

end while

8: a= collect A // n read operations

9: return $\text{sign}(\sum_{j=1}^{n} a[j].sum)$ // return $+1$ or -1, depending on the majority value of the coin flips

Proof: Clearly, true is written to *done* in line 7 only if the process reads at least n^2 flips, therefore $|F_C| \geq n^2$. Consider the point in the execution after n^2 coins were written. Each process can flip at most n more coins until reaching line 7, and then it writes true to *done*. Therefore when true is written to *done* there are at most $2n^2$ coins written, and $|F_C| \leq 2n^2$. ■

For a set of coins F we let $Sum(F)$ be the sum of the coins in F. We denote by F_i the set of coin flips read by the collect of process p_i in Line 8. This is the set according to which the process p_i decides on its output, i.e., p_i returns $Sum(F_i)$. Since each process may flip at most one more coin after true is written to *done*, we can show:

Lemma 4.2 *For every i, $F_C \subseteq F_i$, and $F_i \setminus F_C$ contains at most $n-1$ coins.*

Proof: Note that the collect of any process p_i in Line 8 starts after true is written to *done*. Hence, F_i contains F_C.

After true is written to *done*, each process (except the process that had written true to *done*) can flip at most one more coin before reading that *done* is true in Line 1. Therefore, the set of coins that are written when the process reads Line 8 is the set of coins that were written when true was written to *done* (which is F_C), plus at most $n-1$ additional coins. ■

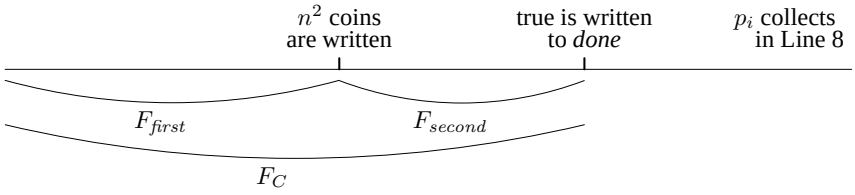


Figure 4.1: Phases of the shared coin algorithm.

We now show that there is at least a constant probability that $Sum(F_C) \geq n$. In this case, by Lemma 4.2, all processes that terminate agree on the value 1, since F_i contains at most $n-1$ additional coins.

We partition the execution into three phases. The first phase ends when n^2 coins are written. After every coin is written there may be up to $n-1$ additional coins that are already flipped but not yet written. The adversary has a choice whether to allow each of these coins to be written. We assume an even stronger adversary that can choose the n^2 written coins out of n^2+n-1 coins that were flipped. The second phase ends when true is written to *done*. In the third phase, each process reads the whole array A and returns a value for the shared coin. (See Figure 4.1.)

Since F_C is the set of coins written when *done* is set to true, then it is exactly the set of coins written in the first and second phases. Let F_{first} be the first n^2 coins that are written, and $F_{second} = F_C \setminus F_{first}$. This implies that $Sum(F_C) = Sum(F_{first}) + Sum(F_{second})$. Therefore, we can bound (from below) the probability that $Sum(F_C) \geq n$ by bounding the probabilities that $Sum(F_{first}) \geq 3n$ and $Sum(F_{second}) \geq -2n$.

Consider the sum of the first n^2+n-1 coin flips. After these coins are flipped, the adversary has to write at least n^2 of them, which will be the coins in F_{first}. If the sum of the first n^2+n-1 coin flips is at least $4n$ then $Sum(F_{first}) \geq 3n$. We bound the probability that this happens using the Central Limit Theorem.

Lemma 4.3 *The probability that $Sum(F_{first}) \geq 3n$ is at least $\frac{1}{8\sqrt{2\pi}}e^{-8}$.*

Proof: There are $N = n^2+n-1$ coins flipped when n^2 coins are written to F_{first}. By the Central Limit Theorem, the probability for the sum of these coins to be at least $x\sqrt{N}$, converges to $1-\Phi(x)$, where $\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{1}{2}y^2}dy$ is the normal distribution function. By [40, Chapter VII], we have $1-\Phi(x) > (\frac{1}{x}-\frac{1}{x^3})\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}x^2}$. Substituting $x=4$ we have that with probability

at least $\frac{1}{8\sqrt{2\pi}}e^{-8}$ the sum of these N coins is at least $4\sqrt{N}$, which is more than $4n$, and hence $Sum(F_{first}) \geq 3n$. ∎

We now need to bound $Sum(F_{second})$. Unlike F_{first}, whose size is determined, the adversary may have control over the number of coins in F_{second}, and not only over which coins are in it. However, by Lemma 4.1 we have $|F_C| \leq 2n^2$, therefore $|F_{second}| \leq n^2$, which implies that F_{second} must be some prefix of n^2 additional coin flips. We consider the partial sums of these n^2 additional coin flips, and show that with high probability, all these partial sums are greater than $-2n$, and therefore in particular $Sum(F_{second}) > -2n$.

Formally, for every i, $1 \leq i \leq n^2$, let X_i be the i-th additional coin flip, and denote $S_j = \sum_{i=1}^{j} X_i$. Since $|F_{second}| \leq n^2$, there exist k, $1 \leq k \leq n^2$, such that $S_k = Sum(F_{second})$. If $S_j > -2n$ for every j, $1 \leq j \leq n^2$, then specifically $Sum(F_{second}) = S_k > -2n$.

The bound on the partial sums is derived using Kolmogorov's inequality.

Kolmogorov's Inequality [40, Chapter IX] *Let $X_1, \ldots X_m$ be independent random variables such that $Var[X_i] < \infty$ for every i, $1 \leq i \leq m$, and let $S_j = \sum_{i=1}^{j} X_i$ for every j, $1 \leq j \leq m$. Then for every $\lambda > 0$, the probability that*

$$|S_j - E[S_j]| < \lambda\sqrt{Var[S_m]} \quad , \quad \textit{for all } j, 1 \leq j \leq m,$$

is at least $1 - \lambda^{-2}$.

Lemma 4.4 *The probability that $S_j > -2n$ for all j, $1 \leq j \leq n^2$, is at least $\frac{3}{4}$.*

Proof: The results of the n^2 coin flips are independent random variables $X_1, \ldots, X_{n^2}$, with $E[X_i] = 0$ and $Var[X_i] = 1$, for every i, $1 \leq i \leq n^2$.

Since S_j is the sum of j independent random variables, its expectation is $E[S_j] = \sum_{i=1}^{j} E[X_i] = 0$, and its variance is $Var[S_j] = \sum_{i=1}^{j} Var[X_i] = j$.

Kolmogorov's inequality implies that $|S_j| < 2n$ (and hence $S_j > -2n$), for all j, $1 \leq j \leq n^2$, with probability at least $\frac{3}{4}$. ∎

This bounds the probability of agreeing on the same value for the shared coin as follows.

Lemma 4.5 *Algorithm 4.1 is a shared coin algorithm with agreement parameter $\delta = \frac{3}{32\sqrt{2\pi}}e^{-8}$.*

Proof: By Lemma 4.3, the probability that $Sum(F_{first}) \geq 3n$ is at least $\frac{1}{8\sqrt{2\pi}}e^{-8}$, and by Lemma 4.4, the probability that $Sum(F_{second}) \geq -2n$ is at least $\frac{3}{4}$. Since $Sum(F_C) = Sum(F_{first}) + Sum(F_{second})$, this implies that the probability that $Sum(F_C) \geq n$ is at least $\frac{3}{32\sqrt{2\pi}}e^{-8}$.

By Lemma 4.2, for every i, $F_i \setminus F_C$ contains at most $n-1$ coins, which implies that if $Sum(F_C) \geq n$ then $Sum(F_i) \geq 1$, and therefore if p_i terminates, then it will decide 1. Hence, with probability at least $\frac{3}{32\sqrt{2\pi}}e^{-8}$, $Sum(F_C) \geq n$ and all processes which terminate agree on the value 1.

By symmetry, all processes that terminate agree on the value -1 with at least the same probability. ■

Clearly, Algorithm 4.1 flips $O(n^2)$ coins. Moreover, all work performed by processes in reading the array A can be attributed to coin flips. This can be used to show that Algorithm 4.1 has $O(n^2)$ total step complexity.

Lemma 4.6 *Algorithm 4.1 has $O(n^2)$ total step complexity.*

Proof: We begin by counting operations that are not part of a collect. There are $O(1)$ such operations per local coin flip, and by Lemmas 4.1 and 4.2 there are at most $2n^2 + n - 1$ local coin flips, implying $O(n^2)$ operations that are not part of a collect.

Each collect performed by p_i in Line 6, can be associated with the n local coins that can be flipped by p_i before it. By Lemma 4.1, there are at most $2n^2$ coins in F_C, i.e. at most $2n^2$ coin flips during the first and second phases of the algorithm. Therefore, during these phases there can be at most $\frac{2n^2}{n} = 2n$ collects performed by processes in Line 6. In the third phase, every process may perform another collect in Line 6, and another collect in Line 8, yielding at most $2n$ additional collects. Thus, there are at most $4n$ collects performed during the algorithm, yielding a total of $O(n^2)$ steps. ■

Using Algorithm 4.1 in the scheme of [12] (see also Algorithm 8.1 in Section 8.1) yields a randomized consensus algorithm. Informally, the scheme uses single-writer registers in which the processes update their preferences (starting with their inputs), and allows the processes to collect the values of these registers in order to detect agreement. If fast processes agree then they

decide their preference and a slow process also adopts it, otherwise a process changes its preference according to the result of the shared coin. Given a shared coin algorithm with agreement parameter δ and step complexity T, the scheme yields a randomized consensus algorithm with $O(\delta^{-1}T)$ expected step complexity. This implies the next theorem:

Theorem 4.7 *There is a randomized consensus algorithm with $O(n^2)$ total step complexity.*

Chapter 5

Combining Shared Coins

In this chapter we show how to combine shared-coin algorithms into a shared coin that integrates their complexity measures. We begin by carefully defining the choices of our strong adversary.

A *configuration* consists of the local states of all processes, and the values of all the registers. Since we consider randomized algorithms with a strong adversary, we partition the configurations into two categories C_{alg} and C_{adv}. In configurations $C \in C_{alg}$ there is a process waiting to flip a local coin, and in configurations $C \in C_{adv}$ all processes are pending to access the shared memory.

For each configuration $C \in C_{alg}$ where p_i is about to flip a local coin there is a fixed probability space for the result of the coin, which we denote by X_i^C. An element $y \in X_i^C$ with probability $\Pr[y]$ represents a possible result of the local coin flip of p_i from the configuration C. If there is more than one process that is waiting to flip a coin in C, then we fix some arbitrary order of flipping the local coins, for example according to the process identifiers. In this case, p_i will be the process with the smallest identifier that is waiting to flip a coin in C. The next process will flip its coin only from the resulting configuration.

We can now define the *strong adversary* formally, as follows. For every configuration $C \in C_{alg}$ the adversary lets a process p_i, with the smallest identifier, flip its local coin. For every configuration $C \in C_{adv}$, the adversary σ picks an arbitrary process to take a step which accesses the shared memory. Having the adversary wait until all processes flip their current coins does not restrict the adversary's power, since any adversary that makes a decision before scheduling some pending coin-flip, can be viewed as one that schedules the pending coin-flip but ignores its outcome until after making that decision.

Algorithm 5.1 Interleaving shared coin algorithms A and B: code for process p_i.

```
1: while true do
2:     take a step in algorithm A
3:     if terminated in A by returning v then return v        // local computation
4:     take a step in algorithm B
5:     if terminated in B by returning v then return v        // local computation
```

5.1 Interleaving shared-coin algorithms

Let A and B be two shared-coin algorithms. Interleaving A and B is done by performing a loop in which the process executes one step of each algorithm (see Algorithm 5.1). When one of the algorithms terminates, returning some value v, the interleaved algorithm terminates as well, and returns the same value v.

We denote by δ_A and δ_B the agreement parameters of algorithm A and algorithm B, respectively.

We next show that the agreement parameter of the interleaved algorithm is the product of the agreement parameters of algorithms A and B. The idea behind the proof is that since different processes may choose a value for the shared coin based on either of the two algorithms, for all processes to agree on some value v we need all processes to agree on v in both algorithms. In order to deduce an agreement parameter which is the product of the two given agreement parameters, we need to show that the executions of the two algorithms are independent, in the sense that the adversary cannot gain any additional power out of running two interleaved algorithms.

In general, it is not obvious that the agreement parameter of the interleaved algorithm is the product of the two given agreement parameters. In each of the two algorithms it is only promised that there is a constant probability that the adversary cannot prevent a certain outcome, but in the interleaved algorithm the adversary does not have to decide in advance which outcome it tries to prevent from a certain algorithm, since it may depend on how the other algorithm proceeds. For example, it suffices for the adversary to have the processes in one algorithm agree on 0 and have the processes in the other algorithm agree on 1.

The first theorem assumes that one of the algorithms always terminates within some fixed bound on the number of steps, and not only with probability 1. We later extend this result to hold

for any pair of shared coin algorithms.

Theorem 5.1 *If algorithm A always terminates within some fixed bound on the number of steps, then the interleaving of algorithms A and B has agreement parameter $\delta \geq \delta_A \cdot \delta_B$.*

Proof: Since algorithm A always terminates within some fixed bound on the number of steps, the interleaved algorithm also terminates within some fixed bound on the number of steps (at most twice as many). We define the probability of reaching agreement on the value v for every configuration C in one of the algorithms, by backwards induction, as follows.

With every configuration C, we associate a value s that is the maximal number of steps taken by all the processes from configuration C, over all possible adversaries and all results of the local coin flips, until they terminate in the interleaved algorithm (by terminating either in A or in B). Since algorithm A always terminates within some fixed bound on the number of steps, s is well defined.

We denote by $C|_A$ the projection of the configuration C on algorithm A. That is, $C|_A$ consists of the local states referring to algorithm A of all processes, and the values of the shared registers of algorithm A. Similarly we denote by $C|_B$ the projection of C on algorithm B.

We denote by S_C the set of adversaries possible from a configuration C. We consider a partition of S_C into S_C^A and S_C^B, which are the set of adversaries from configuration C whose next step is in algorithm A and B, respectively.

We define the probability $\Pr_v[C]$ for agreeing in the interleaved algorithm by induction on s. In a configuration C for which $s = 0$, all processes terminate in the interleaved algorithm, by terminating either in A or in B. We define $\Pr_v[C]$ to be 1 if all the processes decide v, and 0 otherwise. Let C be any other configuration. If $C \in C_{adv}$, then:

$$\Pr_v[C] = \min_{\sigma \in S_C} \Pr_v[C^\sigma],$$

where C^σ is the resulting configuration after scheduling one process, according to the adversary σ, to access the shared memory[1]. If $C \in C_{alg}$, then:

$$\Pr_v[C] = \sum_{y \in X_i^C} \Pr[y] \cdot \Pr_v[C^y],$$

[1]Notice that the minimum of the probabilities over all adversaries in S_C is well defined, since S_C is always finite because there is a fixed bound on the number of steps in algorithm A and a finite number of processes.

where p_i is the process waiting to flip a local coin in the configuration C and C^y is the resulting configuration after the coin is flipped. Notice that for the initial configuration C_I, we have that $\delta = \min_v \Pr_v[C_I]$.

In order to prove the theorem, we use $\Pr_v^A[C]$ (and similarly $\Pr_v^B[C]$) for a configuration C in the interleaved algorithm, which is the probability that starting from C, all the processes that terminate by terminating in algorithm A (algorithm B) agree on the value v.

We define these probabilities formally by induction on s. We only state the definition of $\Pr_v^A[C]$; the definition of $\Pr_v^B[C]$ is analogous. Notice that $\Pr_v^A[C]$ depends only on the projection $C|_A$ of configuration C on algorithm A, and on the possible adversaries in S_C^A. For a configuration C in which $s = 0$, all the processes have terminated in the interleaved algorithm. We define $\Pr_v^A[C]$ to be 1 if all the processes that terminated by terminating in algorithm A agree on the value v, and 0 otherwise (in the latter case there is at least one process that terminated by terminating in algorithm A, but did not decide on the value v).

Let C be any other configuration, i.e., with $s > 0$. If $C \in C_{adv}$, then:

$$\Pr_v^A[C] = \Pr_v^A[C|_A, S_C^A] = \min_{\sigma \in S_C^A} \Pr_v^A[C^\sigma|_A, S_{C^\sigma}^A],$$

where C^σ is the resulting configuration after scheduling one process, according to the adversary σ, to access the shared memory. If $C \in C_{alg}$, then:

$$\Pr_v^A[C] = \Pr_v^A[C|_A, S_C^A] = \sum_{y \in X_i^C} \Pr[y] \cdot \Pr_v^A[C^y|_A, S_{C^y}^A],$$

where p_i is the process waiting to flip a local coin in the configuration C and C^y is the resulting configuration after the coin is flipped.

We now claim that for every configuration C, $\Pr_v[C] \geq \Pr_v^A[C] \cdot \Pr_v^B[C]$; the proof is by induction on s.

Base case: If $s = 0$, then all processes have terminated in the interleaved algorithm. Processes agree on v if and only if every process decides v, whether it terminates in A or in B, therefore

$$\Pr_v[C] = \Pr_v^A[C] \cdot \Pr_v^B[C].$$

Induction step: Assume the claim holds for any configuration C' with at most $s - 1$ steps remaining to termination under any adversary. Let C be a configuration with at most s steps until termination under any adversary. We consider two cases according to the type of C.

If $C \in C_{adv}$, then:

$$\begin{aligned}\Pr_v[C] &= \min_{\sigma \in S_C} \Pr_v[C^\sigma] \\ &= \min\left\{ \min_{\sigma \in S_C^A} \Pr_v[C^\sigma], \min_{\sigma \in S_C^B} \Pr_v[C^\sigma] \right\},\end{aligned}$$

By the induction hypothesis on the configuration C^σ, with one step less to termination, we get:

$$\begin{aligned}\Pr_v[C] &= \min\left\{ \min_{\sigma \in S_C^A} \left(\Pr_v^A[C^\sigma] \cdot \Pr_v^B[C^\sigma]\right), \min_{\sigma \in S_C^B} \left(\Pr_v^A[C^\sigma] \cdot \Pr_v^B[C^\sigma]\right) \right\} \\ &= \min\left\{ \min_{\sigma \in S_C^A} \left(\Pr_v^A[C^\sigma|_A, S_{C^\sigma}^A] \cdot \Pr_v^B[C^\sigma|_B, S_{C^\sigma}^B]\right), \min_{\sigma \in S_C^B} \left(\Pr_v^A[C^\sigma|_A, S_{C^\sigma}^A] \cdot \Pr_v^B[C^\sigma|_B, S_{C^\sigma}^B]\right) \right\}\end{aligned}$$

where the second equality follows by the definition of $\Pr_v^A[C^\sigma]$ and $\Pr_v^B[C^\sigma]$. If the step taken from C by σ is in algorithm A then $C^\sigma|_B = C|_B$. Moreover, $S_{C^\sigma}^B \subseteq S_C^B$, because a process may terminate in algorithm A and be unavailable for scheduling. Therefore we have

$$\Pr_v^B[C^\sigma|_B, S_{C^\sigma}^B] \geq \Pr_v^B[C|_B, S_C^B].$$

Similarly, if the step taken from C by σ is in algorithm B then $\Pr_v^A[C^\sigma|_A, S_{C^\sigma}^A] \geq \Pr_v^A[C|_A, S_C^A]$. Thus,

$$\begin{aligned}\Pr_v[C] &\geq \min\left\{ \min_{\sigma \in S_C^A} \left(\Pr_v^A[C^\sigma|_A, S_{C^\sigma}^A] \cdot \Pr_v^B[C|_B, S_C^B]\right), \min_{\sigma \in S_C^B} \left(\Pr_v^A[C|_A, S_C^A] \cdot \Pr_v^B[C^\sigma|_B, S_{C^\sigma}^B]\right) \right\} \\ &= \min\left\{ \Pr_v^B[C|_B, S_C^B] \left(\min_{\sigma \in S_C^A} \Pr_v^A[C^\sigma|_A, S_{C^\sigma}^A] \right), \Pr_v^A[C|_A, S_C^A] \left(\min_{\sigma \in S_C^B} \Pr_v^B[C^\sigma|_B, S_{C^\sigma}^B] \right) \right\} \\ &= \min\left\{ \Pr_v^B[C] \cdot \Pr_v^A[C], \Pr_v^A[C] \cdot \Pr_v^B[C] \right\} \\ &= \Pr_v^A[C] \cdot \Pr_v^B[C],\end{aligned}$$

which completes the proof of the claim that $\Pr_v[C] \geq \Pr_v^A[C] \cdot \Pr_v^B[C]$ for a configuration $C \in C_{adv}$.

If $C \in C_{alg}$, with process p_i waiting to flip a local coin, then:

$$\begin{aligned}\Pr_v[C] &= \sum_{y \in X_i^C} \Pr[y] \cdot \Pr_v[C^y] \\ &= \sum_{y \in X_i^C} \Pr[y] \cdot \Pr_v^A[C^y] \cdot \Pr_v^B[C^y] \\ &= \sum_{y \in X_i^C} \Pr[y] \cdot \Pr_v^A[C^y|_A, S_{C^y}^A] \cdot \Pr_v^B[C^y|_B, S_{C^y}^B],\end{aligned}$$

by the induction hypothesis on the configuration C^y, with one step less to termination. If the coin of p_i is in algorithm A, then $C^y|_B = C|_B$. Moreover, $S^B_{C^y} \subseteq S^B_C$, because a process may terminate in algorithm A and be unavailable for scheduling. Therefore we have $\Pr^B_v[C^y|_B, S^B_{C^y}] \geq \Pr^B_v[C|_B, S^B_C]$. Similarly, if the coin of p_i is in algorithm B then $\Pr^A_v[C^y|_A, S^A_{C^y}] \geq \Pr^A_v[C|_A, S^A_C]$. Assume, without loss of generality, that the coin is in algorithm A, thus,

$$\begin{aligned}
\Pr_v[C] &\geq \sum_{y \in X_i^C} \Pr[y] \cdot \Pr^A_v[C^y|_A, S^A_{C^y}] \cdot \Pr^B_v[C|_B, S^B_C] \\
&= \Pr^B_v[C|_B, S^B_C] \left(\sum_{y \in X_i^C} \Pr[y] \cdot \Pr^A_v[C^y|_A, S^A_{C^y}] \right) \\
&= \Pr^B_v[C] \cdot \Pr^A_v[C],
\end{aligned}$$

which completes the proof of the claim that $\Pr_v[C] \geq \Pr^A_v[C] \cdot \Pr^B_v[C]$ for a configuration $C \in C_{alg}$.

The theorem follows by applying the claim to the initial configuration C_I, to get that $\Pr_v[C_I] \geq \Pr^A_v[C_I] \cdot \Pr^B_v[C_I]$. Notice again that in the interleaved algorithm, the adversary is slightly more restricted in scheduling processes to take steps in algorithm A than it is in algorithm A itself, since a process might terminate in algorithm B and be unavailable for scheduling. This only reduces the power of the adversary, implying that $\Pr^A_v[C_I] \geq \delta_A$. The same applies for algorithm B and hence $\Pr^B_v[C_I] \geq \delta_B$. Therefore we have that $\Pr_v[C_I] \geq \Pr^A_v[C_I] \cdot \Pr^B_v[C_I] \geq \delta_A \cdot \delta_B$, for every $v \in \{0, 1\}$, which completes the proof since $\delta = \min_v \Pr_v[C_I]$. ∎

When neither algorithm A nor algorithm B have a bound on the number of steps until termination, the same result is obtained by considering *truncated* algorithms. In a truncated algorithm A_h, we stop the original algorithm A after at most h steps, for some finite number h, and if not all processes have terminated then we regard this execution as one that does not agree on any value. This only restricts the algorithm, so the agreement parameter of a truncated algorithm is at most the agreement parameter of the original algorithm, i.e., $\delta_{A_h} \leq \delta_A$.

For any shared coin algorithm, we define $\Pr^h_v[C_I]$ to be the probability that all the processes terminate and decide v within at most h steps from the initial configuration C_I. This is exactly the probability $\Pr_v[C_I]$ of the truncated algorithm A_h. The next lemma proves that the probabilities $\Pr^h_v[C_I]$ tend to the probability $\Pr_v[C_I]$, as h goes to infinity.

Lemma 5.2 *$Pr_v[C_I] = \lim_{h \to \infty} Pr^h_v[C_I]$.*

Proof: For every h, let Y_h be the event that all the processes terminate and decide v within at most h steps from the initial configuration C_I. By this definition, we have $\Pr[Y_h] = \Pr_v^h[C_I]$, and $\Pr[\cup_{h=1}^{\infty} Y_h] = \Pr_v[C_I]$.

It is easy to see that the sequence of events $Y_1, Y_2, \ldots$ is a monotone increasing sequence of events, i.e., $Y_1 \subseteq Y_2 \subseteq \cdots$, therefore the limit $\lim_{h\to\infty} \Pr[Y_h]$ exists, and by [45, Lemma 5, p. 7] we have $\Pr[\cup_{h=1}^{\infty} Y_h] = \lim_{h\to\infty} \Pr[Y_h]$. This implies that $\Pr_v[C_I] = \lim_{h\to\infty} \Pr_v^h[C_I]$. ■

We use the above limit to show that we can truncate an algorithm to get as close as desired to the agreement parameter by a bounded algorithm.

Lemma 5.3 *In a shared coin algorithm with agreement parameter δ, for every $\epsilon > 0$ there is an integer h_ϵ such that $\Pr_v^{h_\epsilon}[C_I] \geq \delta - \epsilon$.*

Proof: Assume, towards a contradiction, that for every h we have $\Pr_v^h[C_I] < \delta - \epsilon$. Since $\Pr_v[C_I]$ is the probability that all the processes terminate and decide v (without a bound on the number of steps), this implies that

$$\Pr_v[C_I] = \lim_{h\to\infty} \Pr_v^h[C_I] \leq \delta - \epsilon < \delta,$$

which completes the proof. ■

We can now truncate the shared coin algorithm A after a finite number of steps as a function of ϵ, and use Theorem 5.1 to get an interleaved algorithm with agreement parameter $\delta_{A_{h_\epsilon}} \cdot \delta_B \geq (\delta_A - \epsilon) \cdot \delta_B$.

By truncating algorithm A we only restrict the interleaved algorithm (as we only restrict algorithm A), and therefore we have that by interleaving algorithms A and B we obtain an agreement parameter δ that is at least $(\delta_A - \epsilon) \cdot \delta_B$ for every $\epsilon > 0$, which implies that $\delta \geq \delta_A \cdot \delta_B$, and gives the following theorem.

Theorem 5.4 *The interleaving of algorithms A and B has agreement parameter $\delta \geq \delta_A \cdot \delta_B$.*

5.2 Applications

5.2.1 Shared Coin Using Single-Writer Registers

We obtain a shared-coin algorithm using only single-writer registers that has both $O(n^2 \log n)$ total work and $O(n \log^2 n)$ individual work, by interleaving the algorithm from [28] and the algorithm from [13].

For two shared-coin algorithms A and B, we denote by $T_A(n)$ and $I_A(n)$ the total and individual work, respectively, of algorithm A, and similarly denote $T_B(n)$ and $I_B(n)$ for algorithm B. We now argue that the total and individual step complexities of the interleaved algorithm are the minima of the respective complexities of algorithms A and B.

Lemma 5.5 *The interleaving of algorithms A and B has an expected total work of*

$$2 \min\{T_A(n), T_B(n)\} + n,$$

and an expected individual work of

$$2 \min\{I_A(n), I_B(n)\} + 1.$$

Proof: We begin by proving the claim regarding the total work. After at most $2T_A(n) + n$ total steps are executed by the adversary, at least $T_A(n)$ of them are in algorithm A, and hence all the processes have terminated in Algorithm A, and have therefore terminated in the interleaved algorithm. The same applies to Algorithm B. Therefore the interleaved algorithm has a total work of $2 \min\{T_A(n), T_B(n)\} + n$.

We now prove the bound on the individual work. Consider any process p_i. After at most $2I_A(n) + 1$ total steps of p_i are executed by the adversary, at least $I_A(n)$ of them are in algorithm A, and hence the process p_i has terminated in Algorithm A, and has therefore terminated in the interleaved algorithm. The same applies to Algorithm B. This is true for all the processes, therefore the interleaved algorithm has an individual work of $2 \min\{T_A(n), T_B(n)\} + 1$. ■

Applying Lemma 5.5 and Theorem 5.1 to an interleaving of the algorithms of [28] and [13], yields:

Theorem 5.6 *There is a shared-coin algorithm with a constant agreement parameter, with $O(n^2 \log n)$ total work and $O(n \log^2 n)$ individual work, using single-writer multi-reader registers.*

5.2.2 Shared Coin for Different Levels of Resilience

In this section we discuss the interleaving done by Saks, Shavit, and Woll [66]. They presented the following three shared-coin algorithms, all having a constant agreement parameter. The complexity measure they consider is of *time units*: one time unit is defined to be a minimal interval in the execution of the algorithm during which each non-faulty processor executes at least one step.

The first algorithm takes $O(\frac{n^3}{n-f})$ time, where f is the number of faulty processes. It is wait-free, i.e., it can tolerate $f = n-1$ faulty processes. This is done by having each process repeatedly flip a local coin and write it into an array, then collect the array to see if at least n^2 coins were already flipped. Once a process observes that enough coins were flipped, it terminates and decides on the majority of all the coins it collected. The individual work of the first algorithm is in $O(n^3)$, since in the worst case, the process does all the work by itself.

The second algorithm takes $O(\log n)$ time in executions with at most $f = \sqrt{n}$ faulty processes, but may not terminate otherwise. This is done by having each process flip one local coin and write it into an array, then repeatedly scan the array until it observes that at least $n - \sqrt{n}$ coins were already flipped. It then terminates and decides on the majority of all the coins it collected.

In the third algorithm there is one predetermined process that flips one local coin and writes the result into a shared memory location. The other processes repeatedly read that location until they see it has been written into, and then decide that value. This takes $O(1)$ time in failure-free executions, but may not terminate otherwise.

Theorem 5.1 shows that the interleaving of these three algorithm gives a shared coin algorithm with a constant agreement parameter. Technically, we apply the theorem twice, first to interleave the first algorithm (which is bounded) and the second algorithm; then we interleave the resulting algorithm (which is bounded) with the third algorithm.

The interleaving of all three algorithms enjoys all of the above complexities, by an argument similar to Lemma 5.5, which yields the following theorem.

Theorem 5.7 *There is a shared-coin algorithm with a constant agreement parameter, which takes $O(\frac{n^3}{n-f})$ time in any execution, $O(\log n)$ time in executions with at most $f = \sqrt{n}$ failures, and $O(1)$ time in failure-free executions, using single-writer multi-reader registers.*

This can of course be further improved by replacing the first algorithm with the algorithm of

Section 5.2.1 or with the algorithm of Aspnes and Censor [11], if multi-writer registers can be employed.

Chapter 6

Polylogarithmic Concurrent Data Structures from Monotone Circuits

All of the known shared coins mentioned in Chapter 2 are essentially based on a majority voting mechanism, which varies in the way the votes are collected and the way termination is detected. The number of votes needed is mainly influenced by the ability of the strong adversary to delay $n-1$ votes in between the collection of votes performed by different processes, since our goal is to provide a large (usually constant) probability of processes seeing the same value of the majority. For example, in the shared coin we present in Chapter 4, n^2 votes are needed.

Detecting that the number of votes reached the desired threshold is a counting problem. The number of steps required for counting the votes induces a tradeoff in the design of shared-coin algorithms: on one hand the processes should not perform the counting procedure very often in order to reduce the step complexity, and on the other hand performing frequent counting allows the majorities seen by different processes to be similar, resulting in a good agreement parameter.

The simple counters used in previous shared coins were of $O(n)$ steps per counter-read operation, which implies that in order to obtain an individual step complexity of $O(n)$ for the shared coin (and hence for randomized consensus) only a constant number of counter-read operations can be invoked by each process. This, however, harms the attempt to have a constant agreement parameter.

Therefore, we are interested in wait-free implementations of counters and additional data structures, in which any operation on the data structure terminates within a sub-linear number of its

steps regardless of the schedule chosen by the adversary, i.e., the *cost* of the implementation is sub-linear.

We note that in this chapter we consider the set of processes to be $P = \{p_0, \ldots, p_{n-1}\}$.

6.1 Max registers

Our basic data structure is a *max register*, which is an object r that supports a `WriteMax`(r, t) operation with an argument t that records the value t in r, and a `ReadMax`(r) operation returning the maximum value written to the object r. A max register may be either bounded or unbounded. For a bounded max register, we assume that the values it stores are in the range $0..(m-1)$, where m is the *size* of the register. We assume that any non-negative integer can be stored in an unbounded max register. In general, we will be interested in unbounded max registers, but will consider bounded max registers in some of our constructions and lower bounds.

One way to implement max registers is by using snapshots. Given a linear-time snapshot algorithm (e.g., [49]), a `WriteMax` operation for process p_i updates location $a[i]$, while a `ReadMax` operation takes a snapshot of all locations and returns the maximum value. Assuming no bounds on the size of snapshot array elements, this gives an implementation of an unbounded max register with linear cost (in the number of processes n) for both `WriteMax` and `ReadMax`. We show below how to build more efficient max registers: a recursive construction that gives costs logarithmic in the size of the register for both `WriteMax` and `ReadMax`.

Note that another approach is to use *f-arrays*, as proposed by Jayanti [50]. An f-array is a data structure that supports computation of a function f over the components of an array. Instead of having a process take a snapshot of the array and then locally apply f to the result, Jayanti implements an f-array by having the write operations update a predetermined location according to the new value of f, which requires a read operation to only read that location. This construction is then extended to a tree algorithm. For implementing an f-array of m registers, where f can be any common aggregate function, including the maximum value or the sum of values, this reduces the number of steps required to $O(\log m)$ for a write operation, while a read operation takes $O(1)$ steps. These implementations use LL/SC objects, while we restrict our base objects to multi-writer multi-reader registers.

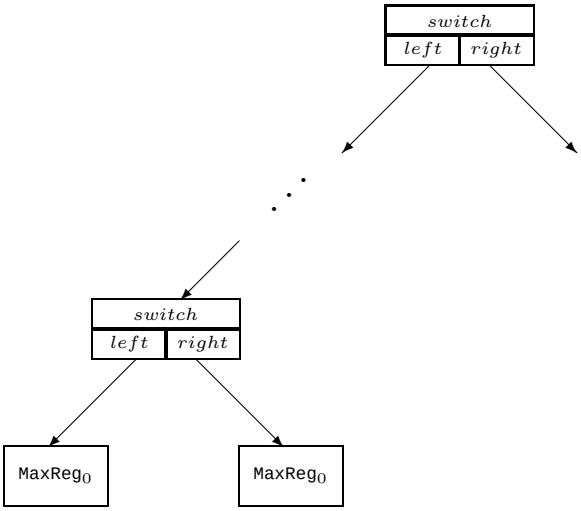


Figure 6.1: Implementing a max register.

In the remainder of this section we show how to construct a max register recursively from a tree of increasingly large max registers. The resulting data structure can also be viewed as a tree whose leaves represent the possible values that can be stored. However, the recursive description facilitates the proof.

The smallest object is a trivial $\texttt{MaxReg}_0$ object, which is a max register r that supports only the value 0. The implementation of $\texttt{MaxReg}_0$ requires zero space and zero step complexity: $\texttt{WriteMax}(r, 0)$ is a no-op, and $\texttt{ReadMax}(r)$ always returns 0.

To get larger max registers, we combine smaller ones recursively (see Figure 6.1). The base objects will consist of at most one snapshot-based max register as described earlier (used to limit the depth of the tree in the unbounded construction) and a large number of trivial $\texttt{MaxReg}_0$ objects. A recursive MaxReg object has three components: two MaxReg objects called r.left and r.right, where r.left is a bounded max register of size m, and one 1-bit multi-writer register called r.switch. The resulting object is a max register whose size is the sum of the sizes of r.left and r.right, or unbounded if r.right is unbounded .

Writing a value t to r is by the $\texttt{WriteMax}(r, t)$ procedure, in which the process writes the value t to r.left if $t < m$ and r.switch is off, or otherwise writes the value $t - m$ to r.right and sets the r.switch bit. Reading the maximal value is by the $\texttt{ReadMax}(r)$ procedure, in which the process returns the value it reads from r.left if r.switch is off, and otherwise returns the value it reads from r.right plus m.

Algorithm 6.1 `WriteMax`(r, t)

Shared variables: $switch$: a 1-bit multi-writer register, initially 0

$left$, a `MaxReg` object of size m, initially 0,

$right$, a `MaxReg` object of arbitrary size, initially 0

```
1: if t < m
2:     if r.switch == 0
3:         WriteMax(r.left, t)
4: else
5:     WriteMax(r.right, t − m)
6:     r.switch = 1
```

Algorithm 6.2 `ReadMax`(r)

Shared Variables: $switch$: a 1-bit multi-writer register, initially 0

$left$, a `MaxReg` object of size m, initially 0,

$right$, a `MaxReg` object of arbitrary size, initially 0

```
1: if r.switch == 0
2:     return ReadMax(r.left)
3: else
4:     return ReadMax(r.right) + m
```

An important property of this implementation is that it preserves linearizability, as shown in the following lemma.

Lemma 6.1 *If* r.left *and* r.right *are linearizable max registers, then so is* r.

Proof: We assume that each of the `MaxReg` objects r.left and r.right is linearizable. Thus, we can associate each operation on them with one linearization point and treat these operations as atomic. In addition, we can associate each read or write to the register r.switch with a single linearization point since it is atomic.

We now consider a schedule of `ReadMax`(r) and `WriteMax`(r, t) operations. These consist of reads and writes to r.switch and of `ReadMax` and `WriteMax` operations on r.left and r.right. We divide the operations on r into three categories:

- C_{left}: `ReadMax`(r) operations that read 0 from r.switch, and `WriteMax`(r, t) operations with $t < m$ that read 0 from r.switch.
- C_{right}: `ReadMax`(r) operations that read 1 from r.switch, and `WriteMax`(r, t) operations with $t \geq m$ (i.e., that write 1 to r.switch).
- C_{switch}: `WriteMax`(r, t) operations with $t < m$ that read 1 from r.switch.

Inspection of the code shows that each operation on r falls into exactly one of these categories. Notice that an operation is in C_{left} if and only if it invokes an operation on r.left, an operation is in C_{right} if and only if it invokes an operation on r.right, and an operation is in C_{switch} if and only if it invokes no operation on r.left or r.right. We order the operations by the following four rules:

1. We order all operations of C_{left} before those of C_{right}. This preserves the execution order of non-overlapping operations between these two categories, since an operation that starts after an operation in C_{right} finishes cannot be in C_{left}.
2. An operation op in C_{switch} is ordered at the latest time possible before any operation op' that starts after op finishes.
3. Within C_{left} we order the operations according to the time at which they access r.left, i.e., by the order of their respective linearization points.
4. Within C_{right} we order the operations according to the time at which they access r.right, i.e., by the order of their linearization points.

It is easy to verify that these rules are well-defined.

We first prove that these rules preserve the execution order of non-overlapping operations. For two operations in the same category this is clearly implied by rules 2–4. Since rule 1 shows that two operations from C_{left} and C_{right} are also properly ordered, it is left to consider the case that one operation is in C_{switch} and the other is either in C_{left} or in C_{right}. In this case, rule 2 implies that their order preserves the execution order.

We now prove that this order satisfies the specification of a max register, i.e., if a `ReadMax`(r) operation op returns t then t is the largest value written by operations on r of type `WriteMax` that are ordered before op. This requires showing that there is a `WriteMax`(r, t) operation op_w

ordered before op, and that there is no $\texttt{WriteMax}(r, t')$ operation $op_{w'}$ with $t' > t$ ordered before op.

This is obtained again by using the linearizability of the components. If op returns a value $t < m$ (i.e., it is in C_{left}) then this is the value that is returned from its invocation op' of $\texttt{ReadMax}(r.\text{left})$. By the linearizability of $r.\text{left}$, there is a $\texttt{WriteMax}(r.\text{left}, t)$ operation op'_w ordered before op' in the linearization of $r.\text{left}$. By rule 3, this implies that the $\texttt{WriteMax}(r, t)$ operation op_w which invoked op'_w is ordered before op. A similar argument for $r.\text{right}$ applies if op returns a value $t \geq m$.

To prove that no operation of type WriteMax with a larger value is ordered before op, we assume, towards a contradiction, that there is a $\texttt{WriteMax}(r, t')$ operation $op_{w'}$ with $t' > t$ that is ordered before op. If op returns a value $t < m$ (i.e., it is in C_{left}) then $op_{w'}$ cannot be in C_{right}, otherwise it would be ordered after op, by rule 1. Moreover, $op_{w'}$ cannot be in C_{switch}, since rule 2 implies that op starts after $op_{w'}$ finishes and hence must also read 1 from $r.\text{switch}$ which is a contradiction to $op \in C_{\text{left}}$. Therefore, $op_w \in C_{\text{left}}$, but this contradicts the linearizability of $r.\text{left}$. If op returns a value $t \geq m$ (i.e., it is in C_{right}) then $op_{w'}$ cannot be in C_{left} because $t' > t$. Moreover, $op_{w'}$ cannot be in C_{switch}, since $t' > t \geq m$. Therefore, op_w is in C_{right}, which contradicts the linearizability of $r.\text{right}$. ■

Using Lemma 6.1, we can build a max register whose structure corresponds to an arbitrary binary search tree, where each internal node of the tree is represented by a recursive max register and each leaf is a $\texttt{MaxReg}_0$, or, for the rightmost leaf, a $\texttt{MaxReg}_0$ or snapshot-based MaxReg depending on whether we want a bounded or an unbounded max register. There are several natural choices, as we will discuss next.

6.1.1 Using a balanced binary search tree

To construct a bounded max register of size 2^k, we use a balanced binary search tree. Let $\texttt{MaxReg}_k$ be a recursive max register built from two $\texttt{MaxReg}_{k-1}$ objects, with $\texttt{MaxReg}_0$ being the trivial max register defined previously. Then $\texttt{MaxReg}_k$ has size 2^k for all k. It is linearizable by induction on k, using Lemma 6.1 for the induction step.

We can also easily compute an exact upper bound on the cost of ReadMax and WriteMax on a $\texttt{MaxReg}_k$ object. For $k = 0$, both ReadMax and WriteMax perform no operations. For larger

k, each `ReadMax` operation performs one register read and then recurses to perform a single `ReadMax` operation on a $\texttt{MaxReg}_{k-1}$ object, while each `WriteMax` performs either a register read or a register write plus at most one recursive call to `WriteMax`. Thus:

Theorem 6.2 *A* $\texttt{MaxReg}_k$ *object implements a linearizable max register for which every* `ReadMax` *operation requires exactly* k *register reads, and every* `WriteMax` *operation requires at most* k *register operations.*

In terms of the size of the max register, operations on a max register that supports m values, where $2^{k-1} < m \leq 2^k$ values, each take at most $\lceil \lg m \rceil$ steps. Note that this cost does not depend on the number of processes n; indeed, it is not hard to see that this implementation works even with infinitely many processes.

6.1.2 Using an unbalanced binary search tree

In order to implement max registers that support unbounded values, we use unbalanced binary search trees.

Bentley and Yao [26] provide several constructions of unbalanced binary search trees with the property that the i-th leaf is at depth $O(\log i)$. The simplest of these, called B_1, constructs the tree by encoding each positive integer using a modified version of a classic variable-length code known as the Elias delta code [38]. In this code, each positive integer $N = 2^k + \ell$ with $0 \leq \ell < 2^k$ is represented by the bit sequence $\delta(N) = 1^{k-1}0\beta(\ell)$, where $\beta(\ell)$ is the $(k-1)$-bit binary expansion of ℓ. The first few such encodings are $0, 100, 101, 11000, 11001, 11010, 11011, 1110000, \ldots$. If we interpret a leading 0 bit as a direction to the left subtree and a leading 1 bit as a direction to the right subtree, this gives a binary tree that consists of an infinitely long rightmost path (corresponding to the increasingly long prefixes 1^k), where the i-th node in this path has a left subtree that is a balanced binary search tree with 2^i leaves. (A similar construction is used in [17].)

Let us consider what happens if we build a max register using the B_1 search tree (see Figure 6.2). A `ReadMax` operation that reads the value v will follow the path corresponding to $\delta(v+1)$, and in fact will read precisely this sequence of bits from the switch registers in each recursive max register along the path. This gives a cost to read value v that is equal to $|\delta(v+1)| = 2\lceil \lg(v+1) \rceil + 1$. Similarly, the cost of $\texttt{WriteMax}(v)$ will be at most $2\lceil \lg(v+1) \rceil + 1$.

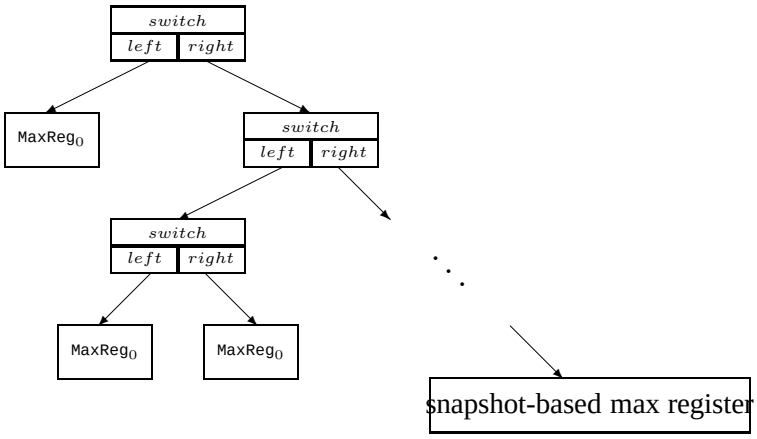


Figure 6.2: An unbalanced max register.

Both of these costs are unbounded for unbounded values of v. For `ReadMax` operations, there is an additional complication: repeated concurrent `WriteMax` operations might set each switch just before the `ReadMax` reaches it, preventing the `ReadMax` from terminating[1]. Another complication is in proving linearizability, as the induction does not bottom without trickery like truncating the structure just below the last node actually used by any completed operation.

For these reasons, we prefer to backstop the tree with a single snapshot-based max register that replaces the entire subtree at position 1^n, where n is the number of processes. Using this construction, we have:

Theorem 6.3 *There is a linearizable implementation of* `MaxReg` *for which every* `ReadMax` *operation that returns value* v *requires* $\min(2\lceil\lg(v+1)\rceil+1, O(n))$ *register reads, and every* `WriteMax` *operation requires at most* $\min(2\lceil\lg(v+1)\rceil+1, O(n))$ *register operations.*

If constant factors are important, the 2 can be reduced to $1+o(1)$ by using a more sophisticated unbalanced search tree; the interested reader should consult [26] for examples.

[1]Note that the infinite-tree construction does give an *obstruction-free* algorithm, where an operation is only required to terminate when running alone.

6.2 Monotone circuits

In this section, we show how a max register can be used to construct more sophisticated data structures from arbitrary monotone circuits. Recall that a monotone circuit computes a function over some finite alphabet of size m, which is assumed to be totally ordered. The circuit is represented by a directed acyclic graph where each node corresponds to a gate that computes a function of the outputs of its predecessors. Nodes with in-degree zero are input nodes; nodes with out-degree zero are output nodes. Each gate g, with k inputs, computes some monotone function f_g of its inputs. Monotonicity means that if $x_i \geq y_i$ for all i, then $f_g(x_1, \ldots, x_k) \geq f_g(y_1, \ldots, y_k)$.

For each monotone circuit, we can construct a corresponding monotone data structure. This data structure supports operations `WriteInput` and `ReadOutput`, where each `WriteInput` operation updates the value of one of the inputs to the circuit and each `ReadOutput` operation returns the value of one of the outputs. Like the circuit as a whole, the effects of `WriteInput` operations are monotone: attempts to set an input to a value less than or equal to its current value have no effect. This restriction still allows for an interesting class of data structures, the most useful of which may be the bounded counter described in Section 6.3.1.

The resulting data structure always provides *monotone consistency*, which is generally weaker than linearizability:

Definition 6.1 *A monotone data structure is* monotone consistent *if the following properties hold in any execution:*

1. *For each output, there is a total ordering $<$ on all* `ReadOutput` *operations for it, such that if some operation R_1 finishes before some other operation R_2 starts, then $R_1 < R_2$, and if $R_1 < R_2$, then the value returned by R_1 is less than or equal to the value returned by R_2.*

2. *The value v returned by any* `ReadOutput` *operation satisfies $f(x_1, \ldots, x_k) \leq v$, where each x_i is the largest value written to input i by a* `WriteInput` *operation that completes before the* `ReadOutput` *operation starts.*

3. *The value v returned by any* `ReadOutput` *operation satisfies $v \leq f(y_1, \ldots, y_k)$, where each y_i is the largest value written to input i by a* `WriteInput` *operation that starts before the* `ReadOutput` *operation completes.*

Algorithm 6.3 `WriteInput`(g, v)

1: `WriteMax`(g, v)
2: Let $g_1, \ldots g_S$ be a topological sort of all gates reachable from g.
3: for $i = 1$ to S
4: `UpdateGate`(g_i)

Algorithm 6.4 `UpdateGate`(g)

1: Let $x_1, \ldots, x_d$ be the inputs to g.
2: for $i = 1$ to d
3: $y_i =$ `ReadMax`(x_i)
4: `WriteMax`$(g, f_g(y_1, \ldots, y_d))$

The intuition here is that the values at each output appear to be non-decreasing over time (the first condition), all completed `WriteInput` operations are always observed by `ReadOutput` (the second condition), and no spurious larger values are observed by `ReadOutput` (the third condition). But when operations are concurrent, it may be that some `ReadOutput` operations return intermediate values that are not consistent with any fixed ordering of `WriteMax` operations, violating linearizability (an example is given in Section 6.3).

We convert a monotone circuit to a monotone data structure by assigning a max register to each input and each gate output in the circuit. We assume that these max registers are initialized to a default minimum value, so that the initial state of the data structure will be consistent with the circuit. A `WriteInput` operation on this data structure updates an input (using `WriteMax`) and propagates the resulting changes through the circuit as described in Procedure `WriteInput`. A `ReadOutput` operation reads the value of some output node, by performing a `ReadMax` operation on the corresponding output. The cost of a `ReadOutput` operation is the same as that of a `ReadMax` operation: $O(\min(\log m, n))$. The cost of `WriteInput` operation depends on the structure of the circuit: in the worst case, it is $O(Sd \min(\log m, n))$, where S is the number of gates reachable from the input and d is the maximum number of inputs to each gate.

Theorem 6.4 *For any fixed monotone circuit C, the* `WriteInput` *and* `ReadOutput` *operations based on that circuit are monotone consistent.*

Algorithm 6.5 `ReadOutput`(g)

1: return `ReadMax`(g)

Proof: Consider some execution of a collection of `WriteInput` and `ReadOutput` operations. We think of this execution as consisting of a sequence of atomic `WriteMax` and `ReadMax` operations and use time to refer to the total number of such operations completed at any point in the execution.

The first clause in Definition 6.1 follows immediately from the linearizability of max registers, since we can just order `ReadOutput` operations by the order of their internal `ReadMax` operations.

For the remaining two clauses, we will jump ahead to the third, upper-bound, clause first. The proof is slightly simpler than the proof for the lower bound, and it allows us to develop tools that we will use for the proof of the second clause.

For each input x_i, let V_i^t be the maximum value written to the register representing x_i at or before time t. For any gate g, let $C_g(x_1, \ldots, x_n)$ be the function giving the output of g when the original circuit C is applied to $x_1, \ldots, x_n$ (see Figure 6.3). For simplicity, we allow C in this case to include internal gates, output gates, and the registers representing inputs (which we can think of as zero-input gates). We thus can define C_g recursively by $C_g(x_1, \ldots, x_n) = x_i$ when $g = x_i$ is an input gate and

$$C_g(x_1, \ldots, x_n) = f_g(C_{g_{i_1}}(x_1, \ldots, x_n), \ldots C_{g_{i_1}}(x_k, \ldots, x_n))$$

when g is an internal or output gate with inputs $g_{i_1} \ldots g_{i_k}$. Let g^t be the actual output of g in our execution at time t, i.e., the contents of the max register representing the output of g. We claim that for all g and t, $g^t \leq C_g(V_1^t, \ldots, V_n^t)$.

The proof is by induction on t and the structure of C. In the initial state, all max registers are at their default minimum value and the induction hypothesis holds. Suppose now that some max register g changes its value at time t. If this max register represents an input, the new value corresponds to some input supplied directly to `WriteInput`, and we have $g^t = C_g(V_1^t, \ldots, V_n^t)$. If the max register represents an internal or output gate, its value is written during some call to `UpdateGate`, and is equal to $f_g(g_{i_1}^{t_1}, g_{i_2}^{t_2}, \ldots, g_{i_k}^{t_k})$ where each g_{i_j} is some register read by this call

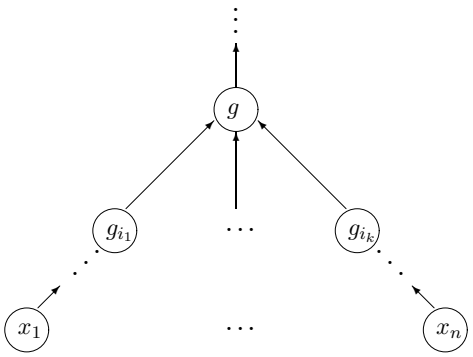


Figure 6.3: A gate g in a circuit computes a function of its inputs $f_g(g_{i_1}, \ldots, g_{i_k})$. The inputs to the circuit are $x_1, \ldots, x_n$.

to `UpdateGate` and $t_j < t$ is the time at which it is read. Because max register values can only increase over time, we have, for each j, $g_{i_j}^{t_j} \leq g_{i_j}^t = g_{i_j}^{t-1} \leq C_{g_{i_j}}(V_1^{t-1}, \ldots, V_n^{t-1})$ by the induction hypothesis, and the fact that only gate g changes at time t. This last quantity is in turn at most $C_{g_{i_j}}(V_1^t, \ldots, V_n^t)$ as only gate g changes at time t. By monotonicity of f_g we then get

$$\begin{aligned} g^t &= f_g(g_{i_1}^{t_1}, g_{i_2}^{t_2}, \ldots, g_{i_k}^{t_k}) \\ &\leq f_g(C_{g_1}(V_1^t, \ldots, V_n^t), \ldots, C_{g_k}(V_1^t, \ldots, V_n^t)) \\ &= C_g(V_1^t, \ldots, V_n^t) \end{aligned}$$

as claimed, which completes the proof of clause 3.

We now consider clause 2, which gives a lower bound on output values. For each time t and input x_i, let v_i^t be the maximum value written to the max register representing x_i by a `WriteInput` operation that finishes at or before time t. We wish to show that for any output gate g, $g^t \geq C_g(v_1^t, \ldots, v_n^t)$. As with the upper bound, we proceed by induction on t and the structure of C. But the induction hypothesis is slightly more complicated, in that in order to make the proof go through we must take into account which gate we are working with when choosing which input values to consider.

For each gate g, let $v_i^t(g)$ be the maximum value written to input register x_i by any instance of `WriteInput` that completes `UpdateGate`(g) at or before time t. Our induction hypothesis is that at each time t and for each gate g, $g^t \geq C_g(v_1^t(g), \ldots v_n^t(g))$. Although in general we have $v_i^t \geq v_i^t(g)$, having $g^t \geq C_g(v_1^t(g), \ldots v_n^t(g))$ implies $g^t \geq C_g(v_1^t, \ldots, v_n^t)$, as any process that

writes to some input x_i that affects the value of g as part of some `WriteInput` operation must complete `UpdateGate`(g) before finishing the operation.

Suppose now that some max register g changes its value at time t. If g is an input, the induction hypothesis holds trivially. Otherwise, consider the set of all `WriteInput` operations that write to g at or before time t. Among these operations, one of them is the last to complete `UpdateGate`(g') for some input g' to g. Let this event occur at time $t' < t$, and call the process that completes this operation p. We now consider the effect of the `UpdateGate`(g) procedure carried out as part of this `WriteInput` operation. Because no other operation completes an `UpdateGate` procedure for any input g_{i_j} to g between t' and t, we have that for each such input and each i, $v_i^t(g_{i_j}) = v_i^{t'}(g_{i_j})$. Since the `ReadMax` operation of each g_{i_j} in p's call to `UpdateGate`(g) occurs after time t', it obtains a value that is at least

$$g_{i_j}^{t'} \geq C_{g_{i_j}}(v_1^{t'}(g_{i_j}), \ldots, v_n^{t'}(g_{i_j})) \geq C_{g_{i_j}}(v_1^t(g_{i_j}), \ldots, v_n^t(g_{i_j})),$$

by the induction hypothesis, monotonicity of $C_{g_{i_j}}$, and the previous observation on the relation between $v_i^{t'}(g_{i_j})$ and $v_i^t(g_{i_j})$. But then

$$\begin{aligned} g^t &\geq f_g(C_{g_{i_1}}(v_1^t(g_{i_1}), \ldots, v_n^t(g_{i_1})), \ldots \\ &\qquad C_{g_{i_k}}(v_1^t(g_{i_k}), \ldots, v_n^t(g_{i_k}))) \\ &\geq f_g(C_{g_{i_1}}(v_1^t(g), \ldots, v_n^t(g)), \ldots, C_{g_{i_k}}(v_1^t(g), \ldots, v_n^t(g))) \\ &= C_g(v_1^t(g), \ldots, v_n^t(g)). \end{aligned}$$

■

6.3 Applications

In this section we consider applications of the circuit-based method for building data structures described in Section 6.2. Most of these applications will be variants on counters, as these are the main example of monotone data structures currently found in the literature. Because we are working over a finite alphabet, all of our counters will be bounded.

The basic structure we will use is a circuit consisting of a binary tree of adders, where each gate in the circuit computes the sum of its inputs and each input to the circuit is assigned to a distinct

process to avoid lost updates. We may consider either bounded or unbounded counters, depending on whether we are using bounded or unbounded max registers. For a bounded counter, we allow only values in the range 0 through $m-1$ for some m; an adder gate whose output would otherwise exceed $m-1$ limits its output to $m-1$. Because the circuit is a tree, a `WriteInput` operation has a particularly simple structure since it need only update gates along a single path to the root; it follows that a `WriteInput` operation costs $O(\min(\log n \log m, n))$ time while a `ReadOutput` operation costs $O(\min(\log m, n))$ time. This is an exponential improvement on the best previously known upper bound of $O(n)$ for exact counting, and on the bound $O(n^{4/5+\epsilon}((1/\delta)\log n)^{O(1/\epsilon)})$, where ϵ is a small constant parameter, for approximate counting which is δ-accurate [11].

If each process is allowed to increase its input by arbitrary values, we get a generalized counter circuit that supports arbitrary non-negative increases to its inputs (the assumption is that each process's input corresponds to the sum of all of its increments so far). Unfortunately, it is not hard to see that the resulting generalized counter is not linearizable, even though it satisfies monotone consistency; the reason is that it may return intermediate values that are not consistent with any ordering of the increments.

Here is a small example of a non-linearizable execution, which we present to illustrate the differences between linearizability and monotone consistency. Consider an execution with three writers, and look at what happens at the top gate in the circuit. Imagine that process p_0 executes a `WriteInput` operation with argument 0, p_1 executes a `WriteInput` operation with argument 1, and p_2 executes a `WriteInput` operation with argument 2. Let p_1 and p_2 arrive at the top gate through different intermediate gates g_1 and g_2; we also assume that each process reads g_2 before g_1 when executing `UpdateGate`(g). Now consider an execution in which p_0 arrives at g first, reading 0 from g_2 just before p_2 writes 2 to g_2. Process p_2 then reads g_2 and g_1 and computes the sum 2 but does not write it yet. Process p_1 now writes 1 to g_1 and p_0 reads it, causing p_0 to compute the sum 1 which it writes to the output gate. Process p_2 now finishes by writing 2 to the output gate. If both these values are observed by readers, we have a non-linearizable schedule, as there is no sequential ordering of the increments 0, 1, and 2 that will yield both output values.

However, for restricted applications, we can obtain a fully linearizable object, as shown in the next subsections.

6.3.1 Linearizable counters with unit increments

Suppose we consider a standard atomic counter object supporting only read and increment operations, where the increment operation increases the value of the counter by exactly one. This is a special case of the generalized counter discussed above, but here the resulting object is linearizable.

To prove linearizability, we consider the counter C as built of a max register at the root output gate g, which adds up two sub-counters, C_1 and C_2, each supporting half of the processes. Our linearizability proof is then by induction, where the base case is a counter for a single process.

Lemma 6.5 *If C_1 and C_2 are linearizable unit-increment counters, then so is C.*

Proof: Each increment operation of C is associated with a value equal to $C_1 + C_2$ at the time it increments C_1 or C_2, considering that C_1 and C_2 are atomic counters according to the induction hypothesis.

An increment operation with an associated value k is linearized at the first time in which a value $\ell \geq k$ is written to the output max register g. A read operation is linearized at the time it reads the output max register g (which we consider to be atomic).

To see that the linearization point for increment k occurs within the interval of the operation, observe that no increment can write a value $\ell \geq k$ to g before increment k finishes incrementing the relevant sub-counter C_1 or C_2, because before then $C_1 + C_2 < k$. Moreover, the increment k cannot finish before $\ell \geq k$ is first written to g, because k writes a value $\ell \geq k$ before it finishes. Since the read operations are also linearized within their execution interval, this order is consistent with the order of non-overlapping operations.

This clearly gives a valid sequential execution, since we now have exactly one increment operation associated with every integer up to any value read from C, and there are exactly k increment operations ordered before a read operation that returns k. ■

Theorem 6.6 *There is an implementation of a linearizable m-valued unit-increment counter of n processes where a read operation takes $O(\min(\log m, n))$ low-level register operations and an increment operation takes $O(\min(\log n \log m, n))$ low-level register operations.*

Proof: Linearizability follows from the preceding argument. For the complexity, observe that the read operation has the same cost as `ReadMax`, while an increment operation requires reading and

updating $O(1)$ max registers per gate at a cost of $O(\min(\log m, 2^i))$ for the i-th gate. The full cost of a write is obtained by summing this quantity as i goes from 0 from $\lceil \lg n \rceil$. ■

Note that for a polynomial number of increments, an increment takes $O(\log^2 n)$ steps. It is also possible to use unbounded max registers, in which case the value m in the cost of a read or increment is replaced by the current value of the counter.

6.3.2 Threshold objects

Another variant of a shared counter that is linearizable is a *threshold object*. This counter allows increment operations, and supports a read operation that returns a binary value indicating whether a predetermined threshold has been crossed. We implement a threshold object with threshold T by having increment operations act as in the generalized counter, and having a read operation return 1 if the value it reads from the output gate is at least T, and 0 otherwise. We show that this implementation is linearizable even with non-uniform increments, where the requirement is that a read operation returns 1 if and only if the sum of the increment operations linearized before it is at least T.

Lemma 6.7 *The implementation of a threshold object C with threshold T by a monotone data structure with the procedures* `WriteInput` *and* `ReadOutput` *is linearizable.*

Proof: We use monotone consistency to prove linearizability for the threshold object C. Let C_1 and C_2 be the sub-counters that are added to the final output gate g.

We order read operations according to the ordering implied by monotone consistency, which is consistent with the order of non-overlapping read operations, and implies that once a read operation returns 1 then any following read operation returns 1. We order write operations according to their execution order, which is clearly consistent with the order of non-overlapping write operations. We then interleave these orders according to the execution order of reads and writes, which implies that this order is consistent with the order of non-overlapping read and write operations.

The interleaving is done while making sure that the sum of increments that are ordered before any read that returns 0 is less than T, and that the sum of increments that are ordered before the first read that returns 1 is at least T. Monotone consistency guarantees that we can do this. For a read operation that returns 0, the value read in g is less than T, therefore the second clause of

monotone consistency implies that the sum of all writes that finish before the read starts is less than T. For a read operation that returns 1, the value read in g is at least T, therefore the third clause implies that there enough increment operations that start before this read finishes that have a sum at least T. ■

Our proof of Lemma 6.7 does not use the specification of a threshold object, but rather the fact that it is an implementation of a monotone circuit with a binary output. We therefore have:

Lemma 6.8 *The implementation of any monotone circuit with a binary output by a monotone data structure with the procedures* `WriteInput` *and* `ReadOutput` *is linearizable.*

Note that for any binary-output circuit, we can represent the output using a 1-bit flag initialized to 0 and set to 1 by any `WriteInput` operation that produces 1 as output (essentially, we use the flag as a 2-valued bounded max register). A reader may then do only one operation which accesses that flag and returns its value.

Chapter 7

Randomized Consensus with Optimal Individual Work

In this section we describe an application of our sub-linear counter algorithm: an algorithm for solving randomized consensus with optimal $O(n)$ work per process. This improves the best previously known bound to match the $\Omega(n)$ lower bound that follows from the result of Chapter 11. While the latter result showed a tight bound of $\Theta(n^2)$ on the *total* number of operations carried out by all processes, the algorithm presented in this chapter guarantees that this work is in fact evenly distributed among all the processes.

As in Chapter 4, we use the standard reduction [12] of randomized consensus to the problem of implementing a shared coin. The code for each process's actions in the shared coin implementation is given as Algorithm 7.1, in which each process outputs either +1 or -1.

We now give a high-level description of the shared coin algorithm, which will be followed by a formal proof. Each process generates votes whose sum is recorded in an array of n single-writer registers, and whose variance is recorded in $2 \log n$ counters. A process terminates and outputs the majority of votes when the total variance of the votes reaches a certain threshold, which is small enough to guarantee the claimed step complexity, and, at the same time, large enough to have a good probability for the votes to have a distinct majority.

In order to reduce the individual step complexity, the votes generated by a process have increasing weights. This allows fast processes running alone to cast heavier votes and reach the variance threshold after generating fewer votes.

Algorithm 7.1 Shared coin algorithm with $O(n)$ individual work.

shared data: array counters$[0..(2\log n)]$ of counters

array votes$[1..n]$ of single-writer registers

multi-writer bit done

1: $i = 0$

2: $v_0 = 0$

3: varianceWritten $= 0$

4: while $v_i < 1$ and not done do

5: $\quad i = i + 1$

6: $\quad w_i = \min\left(\max(v_{i-1}, 1/n), 1/\sqrt{n}\right)$

7: $\quad v_i = v_{i-1} + w_i^2$

8: $\quad$ vote $=$ `LocalCoin()` $\cdot w_i$

9: $\quad$ votes[pid] $=$ votes[pid] $+$ vote

10: $\quad$ if $v_i \geq 2^{\text{varianceWritten}}/n^2$ then

11: $\quad\quad$ `CounterIncrement(`counters[varianceWritten]`)`

12: $\quad\quad$ varianceWritten $=$ varianceWritten $+ 1$

13: $\quad\quad$ if $\sum_{k=0}^{2\log n}\left(2^k \cdot$ `ReadCounter`$(\text{counters}[k])\right) \geq 3n^2$ then

14: $\quad\quad\quad$ break

15: done $=$ **true**

16: return $\operatorname{sgn}(\sum_p \text{votes}[p])$

The weight w_i of the i-th vote is a function of the total variance v_{i-1} of all previous votes, as computed in Line 6; we discuss the choice of this formula in more detail in Section 7.1. The voting operation consists of lines 6 through 9; each time the process votes, it computes the weight w_i of the next vote, updates the total variance v_i, generates a random vote with value $\pm w_i$ with equal probability, and adds this vote to the pool votes[pid], where pid is the current process id.

Termination can occur in one of three ways:

1. The process by itself produces enough variance to cross the threshold (first clause of while loop test in Line 4).
2. All processes collectively produce enough variance for the threshold test to succeed (Line 13).

3. The process observes that some other process has written done (second clause of while loop test in Line 4). This last case can only occur if some other process previously observed sufficient total variance to finish.

We use $2 \log n$ counters, since our counters can be incremented at most once by each process. Having sub-linear counters allows incrementing and reading them not very frequently, namely, only when increasing amounts of variance are generated by the process, which gives the linear complexity.

The counters give the total variance, which when large enough has constant probability for the votes having a distinct majority, even in spite of small differences between the votes that different processes read, which may be caused by the asynchrony of the system.

The proof of correctness for the shared coin algorithm proceeds in several steps. In Section 7.1 we prove some properties of the weight function. These will allow us to bound the expected individual work of each process, and later will also be used to analyze the agreement parameter. In Section 7.2 we bound the individual work (Lemma 7.3), and prove bounds on the probabilities of terminating with a total variance of votes which is too low or too high. Finally, in Section 7.3 we analyze the sums of votes in different phases of Algorithm 7.1, which allows us to prove in Theorem 7.10 that it implements a shared coin with a constant agreement parameter.

7.1 Properties of the weight function

The weight of the i-th vote is given by the formula $w_i = \min\left(\max(v_{i-1}, 1/n), 1/\sqrt{n}\right)$, where $v_{i-1} = \sum_{j=1}^{i-1} w_j^2$ is the total variance contributed by all previous votes.

The cap of $1/\sqrt{n}$ keeps any single vote from being too large, which will help us show in Section 7.3 that the core votes are normally distributed in the limit. The use of $\max(v_{i-1}, 1/n)$ bounds the weight of all unwritten votes in any state by the total variance of all written votes, plus a small constant corresponding to those processes that are still casting the minimum votes of weight $1/n$. This gives a bound on the *bias* that the adversary can create by selectively stopping a process after it generates its i-th vote in Line 8 but before it writes it in Line 9.

Lemma 7.1 *For any values $i_j \geq 0$, we have $\sum_{j=1}^{n} w_{i_j} \leq 1 + \sum_{j=1}^{n} v_{i_j - 1}$.*

Proof: This follows from the assignment in Line 6, by summing w_{i_j} over all j:
$\sum_{j=1}^{n} w_{i_j} \leq \sum_{j=1}^{n} \max(v_{i_j-1}, 1/n) \leq \sum_{j=1}^{n} \left(v_{i_j-1} + 1/n\right) = 1 + \sum_{j=1}^{n} v_{i_j-1}$ ■

Despite wanting to keep w_i small relative to v_i, we still want to generate variance quickly. The following lemma states that any single process can generate a total variance $v_i \geq 1$ after only $i = 4n$ votes. It follows immediately that the loop in Algorithm 7.1 is executed at most $4n$ times.

Lemma 7.2 *All of the following conditions hold:*

1. $v_1 = 1/n^2$
2. $v_{i+1} \leq 2v_i \qquad [i \geq 1]$
3. $v_{4n} \geq 1$.

Proof: We observe that the following recurrence holds for v_i:

$$v_i = v_{i-1} + w_i^2 = v_{i-1} + \left(\min(\max(v_{i-1}, 1/n), 1/\sqrt{n})\right)^2,$$

with a base case of $v_0 = 0$. We can immediately compute $v_1 = 1/n^2$, giving (1).

It also follows that $v_i \geq v_1 = 1/n^2$ for all $i \geq 1$. Let $i \geq 1$ and consider the possible values of v_{i-1}. If $v_{i-1} \leq 1/n$ then $w_i^2 = 1/n^2$, therefore $v_i = v_{i-1} + 1/n^2 \leq 2v_{i-1}$. Otherwise, if $1/n \leq v_{i-1} < 1/\sqrt{n}$ then $w_i^2 = v_{i-1}^2 < 1/n$, therefore $v_i \leq v_{i-1} + 1/n \leq 2v_{i-1}$. Finally, if $v_{i-1} \geq 1/\sqrt{n}$ then $w_i^2 = 1/n$ and we have $v_i = v_{i-1} + 1/n \leq 2v_{i-1}$. So (2) holds for all $i \geq 1$.

To prove (3), we consider three phases of the increase in v_i, depending on whether $w_i = 1/n$, $w_i = v_{i-1} \geq 1/n$, or $w_i = 1/\sqrt{n}$.

In the first phase, we have that for any $i > 0$, $v_i \geq v_{i-1} + 1/n^2$, and thus $v_i \geq i/n^2$. In particular, for $i = n$ we have $v_i \geq 1/n$.

For the second phase, suppose that $v_{i-1} \leq 1/\sqrt{n}$. We then have $v_i \geq v_{i-1} + v_{i-1}^2$. If this holds, and there is some $x \geq 1$ such that $v_i \geq 1/x$, then

$$\begin{aligned} v_{i+1} \geq v_i + v_i^2 \geq \frac{1}{x} + 1/x^2 = \frac{x+1}{x^2} = \frac{(x+1)(x-1/2)}{x^2(x-1/2)} \\ = \frac{x^2 + x/2 - 1/2}{x^2(x-1/2)} = \frac{1 + 1/(2x) - 1/(2x^2)}{x - 1/2} \geq \frac{1}{x-1/2}. \end{aligned}$$

By iterating this calculation, we obtain that $v_{i+t} \geq \frac{1}{x-t/2}$, so long as $v_{i+t-1} \leq 1/\sqrt{n}$. Starting with $v_n \geq 1/n$, we thus get $v_{n+t} \geq 1/(n-t/2)$, which gives $v_i \geq 1/\sqrt{n}$ for some $i \leq (n + (2n - \sqrt{n})) \leq 3n$.

At this point, w_i is capped by $1/\sqrt{n}$; the increment to v_i is thus $w_i^2 = 1/n$, so after a further $(n - \sqrt{n}) \leq n$ votes, we have $v_i \geq 1$. The total number of votes is bounded by $4n$, as claimed. ■

7.2 Termination

We begin analyzing the situation of termination, i.e., when no more votes are generated, by bounding the running time of the algorithm.

Lemma 7.3 *Algorithm 7.1 executes $O(n)$ local coin-flips and $O(n)$ register operations, including those incurred by* `ReadCounter` *operations on the counters.*

Proof: Lemma 7.2 implies that each process terminates after casting at most $4n$ votes. This gives an $O(n)$ bound on the number of iterations of the main loop. Each iteration requires one call to `LocalCoin` and two register operations (the read of done in Line 4 and the write to votes[pid] in Line 9, assuming the previous value of votes[pid] is cached in an internal variable), plus whatever operations are needed to execute the threshold test in Lines 11 through 13. These lines are executed at most $1 + 2\log n$ times (since varianceWritten rises by 1 for each execution), and their cost is dominated by the $1 + 2\log n$ calls to `ReadCounter` at a cost of $O(\text{polylog}n)$ each. The cost of the at most $(1 + 2\log n)^2$ total calls to `ReadCounter` is thus bounded by $O(n)$. ■

Consider the sequence of votes generated by all processes, ordered by the interleaving of execution of the `LocalCoin` procedure. Write X_t for the random variable representing the value of the t-th such vote (or 0 if there are fewer than t total votes); we thus have a sequence of votes $X_1, X_2, \ldots$.

We wish to bound any sum computed in Line 13 according to the total variance of the votes that have been generated, where for a given number of votes t their total variance is $\sum_{i=1}^{t} X_i^2$.

For a given t, consider the state of the counters when the t-th vote is generated. For each process j, let k_j^t be the maximum index of any counter in counters for which j has completed a `CounterIncrement` operation, and let ℓ_j^t be the maximum index of any counter for which j has

started a `CounterIncrement` operation. If there is no such index, set k_j^t or ℓ_j^t to -1. Let i_j^t be the total number of votes generated by process j among the first t votes, i.e., $\sum_{i=1}^{t} X_i^2 = \sum_{j=1}^{n} v_{i_j^t}$. We first bound k_j^t and ℓ_j^t in terms of $v_{i_j^t}$.

Lemma 7.4 *For every t, we have $2^{\ell_j^t} \leq v_{i_j^t} n^2 + 1/2$ and $2^{k_j^t+1} \geq v_{i_j^t} n^2$.*

Proof: We begin with an upper bound on $2^{\ell_j^t}$. Observe that the test in Line 10 means that `CounterIncrement`(counters$[k]$) can have started only if $v_{i_j^t} \geq 2^k/n^2$; it follows that either $\ell_j^t = -1$ or $2^{\ell_j^t} \leq v_{i_j^t} n^2$; in either case we have

$$2^{\ell_j^t} \leq v_{i_j^t} n^2 + 1/2.$$

Getting a lower bound on $2^{k_j^t}$ is slightly harder, since we can't rely solely on the test in Line 10 succeeding but must also show that varianceWritten is large enough that $2^{\text{varianceWritten}}$ is in fact close to v_i^2. We do so by proving, by induction on i, that at the end of each iteration of the main loop in Algorithm 7.1, $v_i \leq 2^{\text{varianceWritten}}/n^2$. To avoid ambiguity (and excessive text), we will write W_i for the value of varianceWritten at the end of the i-th iteration.

The base case is $i = 1$, where inspection of the code reveals $v_1 = 1/n^2$ and $W_1 = 1$; in this case $v_1 \leq 2^{W_1}/n^2 = 2/n^2$. For larger i, suppose that it holds that $v_{i-1} \leq 2^{W_{i-1}}/n^2$. Then $v_i \leq 2v_{i-1} \leq 2^{W_{i-1}+1}/n^2$ (the first inequality follows from (2) of Lemma 7.2). It is possible that v_i is much smaller than this bound, indeed, small enough that $v_i < 2^{W_{i-1}}/n^2$; in this case $W_i = W_{i-1}$ and the invariant continues to hold. If not, Line 12 is executed, and so we have $W_i = W_{i-1} + 1$. But then $v_i \leq 2^{W_{i-1}+1}/n^2 = 2^{W_i}/n^2$, so the invariant holds here as well.

In bounding $2^{k_j^t}$, the worst case (for $k_j^t \geq 0$) is when $k_j^t = W_{i_j^t - 1}$, the value of varianceWritten at the end of the previous iteration of the loop. In this case we have $v_{i_j^t} \leq 2v_{i_j^t - 1} \leq 2 \cdot 2^{k_j^t}/n^2 = 2^{k_j^t+1}/n^2$. For $k_j^t = -1$, we have $i_j^t \leq 1$, so $v_{i_j^t} \leq 1/n^2 = 2^{-1+1}/n^2 = 2^{k_j^t+1}/n^2$. In either case we get

$$2^{k_j^t+1} \geq v_{i_j^t} n^2.$$

■

We now consider the interaction between `CounterIncrement` and `ReadCounter` operations in order to bound any sum computed in Line 13. The next lemma shows a small upper bound on the probability that the sum is too large.

Lemma 7.5 *If S is a sum computed in Line 13, where the first* `ReadCounter` *operation is started after t total votes are generated, then*

$$S \geq n^2 \sum_{i=1}^{t} X_i^2 - n.$$

Proof: For each k let $r[k]$ be the value returned by the `ReadCounter`(counters$[k]$) operation included in the sum, and let $c[k]$ be the number of calls to `CounterIncrement`(counters$[k]$) that have finished before the summation starts. Then $r[k] \geq c[k]$ for every k, which implies that

$$\begin{aligned} S = \sum_{k=0}^{2\log n} 2^k r[k] \geq \sum_{k=0}^{2\log n} 2^k c[k] &= \sum_{j=1}^{n} \sum_{m=0}^{k_j^t} 2^m = \sum_{j=1}^{n} \left(2^{k_j^t+1} - 1\right) \\ &\geq \sum_{j=1}^{n} v_{i_j^t} n^2 - n = n^2 \sum_{i=1}^{t} X_i^2 - n, \end{aligned}$$

where the fourth inequality follows from Lemma 7.4. This completes the proof. ■

Similarly, the next lemma shows a small upper bound on the probability that the sum is too small.

Lemma 7.6 *If S' is a sum computed in Line 13, where the last* `ReadCounter` *operation is completed before t' total votes are generated, then*

$$S' \leq 2n^2 \sum_{i=1}^{t'} X_i^2.$$

Proof: For each k let $r'[k]$ be the value returned by the `ReadCounter`(counters$[k]$) operation included in the sum, and let $c'[k]$ be the number that start before the summation finishes. Then $r'[k] \leq c'[k]$ for every k, which implies that

$$\begin{aligned} S' = \sum_{k=0}^{2\log n} 2^k r'[k] \leq \sum_{k=0}^{2\log n} 2^k c'[k] &= \sum_{j=1}^{n} \sum_{m=0}^{\ell_j^{t'}} 2^m = \sum_{j=1}^{n} \left(2^{\ell_j^{t'}+1} - 1\right) \\ &\leq \sum_{j=1}^{n} \left(2\left(v_{i_j^{t'}} n^2 + 1/2\right) - 1\right) = 2\sum_{j=1}^{n} v_{i_j^{t'}} n^2 = 2n^2 \sum_{i=1}^{t'} X_i^2. \end{aligned}$$

where the fourth inequality follows from Lemma 7.4. This completes the proof. ■

Using the two previous lemmas, we now prove upper and lower bounds on the total variance of all the generated votes.

Lemma 7.7 *Let T be the total number of votes generated by all processes during an execution of the shared coin algorithm, and let $V = \sum_{i=1}^{T} X_i^2$ be the total variance of these votes. Then we have $1 < V < 7 + \frac{4}{n}$.*

Proof: Termination with $V < 1$ cannot occur as the result of some process failing the main loop test $v_i < 1$, as if this test fails, that process alone gives $V \geq 1$. So the only possibility is that the threshold test in Line 13 succeeds for some process despite the low total variance. But since the total variance of all votes is less than 1, for any particular sum of observed counter values S' we have from Lemma 7.6 that $S' \leq 2n^2$ and so termination cannot occur.

For the upper bound on V, suppose that after t_1 votes we have $\sum_{i=1}^{t_1} X_i^2 \geq 3 + 1/n$. If there is no such t_1, then $V < 7 + \frac{4}{n}$; otherwise, let t_1 be the smallest value with this property. Because t_1 is least, we have $\sum_{i=1}^{t_1} X_i^2 < 3 + 1/n + X_{t_1}^2 \leq 3 + 2/n$.

From Lemma 7.5 we have that, for any execution of Line 13 that starts after these t_1 votes, the return value S satisfies $S \geq n^2(3 + 1/n) - n \geq 3n^2$.

This implies every process that executes the threshold test after t_1 total votes will succeed, and as a result will cast no more votes. So we must bound the amount of additional variance each process can add before it reaches this point. Recall that $i_j^{t_1}$ is the number of votes cast by process j among the first t_1 votes, and let i'_j be the total number of votes cast by process j before termination. Then under the assumption that j's next threshold test succeeds, we have $v_{i'_j} < 2v_{i_j^{t_1}} + 1/n$, as j can at most double its variance and cast one additional vote before seeing $v_i \geq 2^{\text{varianceWritten}}$. So now we have

$$V = \sum_{i=1}^{T} X_i^2 = \sum_{j=1}^{n} v_{i'_j} < \sum_{j=1}^{n} (2v_{i_j^{t_1}} + 1/n) = 1 + 2\sum_{j=1}^{n} v_{i_j^{t_1}}$$
$$= 1 + 2\sum_{i=1}^{t_1} X_i^2 < 1 + 2(3 + 2/n) = 7 + \frac{4}{n}.$$

Thus the upper bound on V holds. ∎

7.3 Core votes and extra votes

We will assume for convenience that the adversary scheduler is deterministic, in particular that the choice of which process generates vote X_t is completely determined by the outcomes of votes X_1 through X_{t-1}; this assumption does not constrain the adversary's behavior, because any randomized adversary strategy can be expressed as a weighted average of deterministic strategies. Under this assumption, we have that the weight $|X_t|$ of X_t is a constant conditioned on $X_1 \ldots X_{t-1}$, but because the adversary cannot predict the outcome of `LocalCoin`, the expectation of X_t is zero even conditioning on the previous votes. That $\mathrm{E}[X_t = 0 | X_1, \ldots X_{t-1}]$ is the defining property of a class of stochastic processes known as *martingales* (see [3, 45, 46]); in particular the X_t variables form a *martingale difference sequence* while the variables $S_t = \sum_{i=1}^{t} X_t$ form a martingale proper.

Martingales are a useful class of stochastic processes because for many purposes they act like sums of independent random variables: there is an analog of the Central Limit Theorem that holds for martingales [46, Theorem 3.2], which we use in the proof of Lemma 7.8; and as with independent variables, the variance of S_t is equal to the sum of the variances of X_1 through X_t [46, p. 8], a fact we use in the proof of Lemma 7.9.

Martingales can also be neatly sliced by *stopping times*, where a stopping time is a random variable τ which is finite with probability 1 and for which the event $[\tau \leq t]$ can be determined by observing only the values of X_1 through X_t (see [45, Section 12.4]); the process $\{S'_t = \sum_{i=1}^{t} X'_i\}$ obtained by replacing X_t with $X'_t = X_t$ for $t \leq \tau$ and 0 otherwise, is also a martingale [45, Theorem 12.4.5], as is the sequence $S''_t = \sum_{i=1}^{t} X_{\tau+i}$ [45, Theorem 12.4.11]. We will use a stopping time to distinguish the core and extra votes.

Define τ as the least value such that either (a) $\sum_{t=1}^{\tau} X_t^2 \geq 1$ or (b) the algorithm terminates after τ votes. Observe that τ is always finite, because if the algorithm does not otherwise terminate, any process eventually generates 1 unit of variance on its own (as shown in Lemma 7.2). Because the weights of votes vary, τ is in general a random variable; but for a fixed adversary strategy, the condition $\tau = t$ can be detected by observing the values of $X_1 \ldots X_t$. Thus τ is a stopping time relative to the X_t. The quantity S_τ will be called the *core vote* of the algorithm. The remaining votes $X_{\tau+1}, X_{\tau+2}, \ldots$ form the *extra votes*.

First, we show a constant probability of the core vote being at least a constant. This will follow by an application of the martingale Central Limit Theorem, particularly in the form of Theorem 3.2

from [46]. This theorem considers a zero-mean *martingale array*, which is a sequence of tuples $\{S_{mt}, \mathcal{F}_{mt}, 1 \leq t \leq k_m, m \geq 1\}$ parameterized by m, where for each fixed m the sequence of random variables $\{S_{mt}\}$ is a zero-mean martingale with respect to the corresponding sequence of σ-algebras $\{\mathcal{F}_{mt}\}$, with difference sequence $X_{mt} = S_{mt} - S_{m,t-1}$. Specializing the theorem slightly, if it holds that:

1. $\max_t |X_{mt}| \xrightarrow{p} 0$,
2. $\sum_t X_{mt}^2 \xrightarrow{p} 1$,
3. $\mathrm{E}[\max_t X_{mt}^2]$ is bounded in m, and
4. $\mathcal{F}_{m,t} \subseteq \mathcal{F}_{m+1,t}$ for $1 \leq t \leq k_m, m \geq 1$,

then $S_{mt} \xrightarrow{d} N(0,1)$, where $N(0,1)$ has a normal distribution with zero mean and unit variance. Here $\xrightarrow{p}$ denotes convergence in probability and $\xrightarrow{d}$ denotes convergence in distribution.

Lemma 7.8 *For any fixed α and n sufficiently large, there is a constant probability p_α such that, for any adversary strategy, $\Pr[S_\tau \geq \alpha] \geq p_\alpha$.*

Proof: We construct our martingale array by considering, for each number of processes n, the set of all deterministic adversary strategies for scheduling Algorithm 7.1. The first rows of the array correspond to all strategies for $n = 1$ (in any fixed order); subsequent rows hold all strategies for $n = 2$, $n = 3$, and so forth. Because each set of strategies is finite (for an execution with n process, each choice of the adversary chooses one of n processes to execute the next coin-flip in response to some particular pattern of $O(n^2)$ preceding coin-flips, giving at most $n^{O(n^2)}$ possible strategies), every adversary strategy eventually appears as some row m in the array. We will write n_m as the value of n corresponding to this row and observe that it grows without bound.

For each row in the array, we set k_m to include all possible votes, but truncate the actual set of coin-flips at time τ. Formally, we define $X_{mt} = X_t$ for $t \leq \tau$, but set $X_{mt} = 0$ for larger t. This ensures that $S_{mk_m} = S_\tau$, the total core vote from each execution, while maintaining the martingale property and the fixed-length rows required by the theorem. We ensure the nesting condition (4) by using the same random variable to set the sign of each vote at time t in each row; in effect, we imagine that we are carrying out an infinite collection of simultaneous executions for different values of n and different adversary strategies using the same sequence of random local coin-flips.

We now show the remaining requirements of the theorem hold. For (1), we have that $\max_t |X_{mt}| \leq 1/\sqrt{n_m}$, which converges to 0 absolutely (and thus in probability as well). For (2), by construction of τ and Lemma 7.7, we have that $1 \leq \sum_t X_{mt}^2 \leq 1 + X_{m\tau}^2 \leq 1 + 1/n_m$. Thus $\sum_t X_{mt}^2$ converges in probability to 1. For (3), we again use the fact that $X_{mt}^2 \leq 1/n_m$ for all t.

It follows that S_{mt} converges in distribution to $N(0,1)$. In particular, for any fixed α, we have that $\lim_{m\to\infty} \Pr[S_{mt} \geq \alpha] = \Pr[N(0,1) \geq \alpha]$, which is a constant. By choosing p_α strictly less than this constant, we have that for sufficiently large m (and thus for sufficiently large $n = n_m$), $\Pr[S_\tau \geq \alpha] = \Pr[S_{mt} \geq \alpha] \geq p_\alpha$. ■

By symmetry, we also have $\Pr[S_\tau \leq -\alpha] \geq p_\alpha$.

We now consider the effect of the extra votes. Our goal is to bound the probability that the total extra vote is too large using Chebyshev's inequality, obtaining a bound on the variance of the extra votes from a bound on the sum of the squares of the weights of all votes as shown in Lemma 7.7.

Lemma 7.9 *Define τ' to be the maximum index such that (a) $X_{\tau'} \neq 0$ and (b) $\sum_{i=1}^{\tau'} X_i^2 \leq 7 + 4/n$. Let p_{13} be the probability from Lemma 7.8 that S_τ is at least 13. Then for sufficiently large n and any adversary strategy, $\Pr[S_{\tau'} > 9] \geq (1/8)p_{13}$.*

Proof: From Lemma 7.8, the probability that the sum of the core votes S_τ is at least 13 is at least p_{13}. We wish to show that, conditioning on this event occurring, adding the extra votes up to τ' does not drive this total below 9.

Observe that τ' is a stopping time. For the rest of the proof, all probabilistic statements are conditioned on the values of $X_1 \ldots X_\tau$.

Define $Y_i = X_{\tau+i}$ for $\tau + i \leq \tau'$ and 0 otherwise. Let $U_i = \sum_{j=1}^{i} Y_j$. Then $\{U_i\}$ is a martingale and $\mathrm{E}[U_i] = 0$ for all i. Let $i_{\max}$ be such that $Y_i = 0$ for $i > i_{\max}$ with probability 1 ($i_{\max}$ exists by Lemma 7.3). Then

$$\begin{aligned}\mathrm{Var}[U_{i_{\max}}] = \mathrm{Var}\left[\sum_{i=1}^{i_{\max}} Y_i\right] = \sum_{i=1}^{i_{\max}} \mathrm{Var}\left[Y_i\right] = \sum_{i=1}^{i_{\max}} \mathrm{E}\left[Y_i^2\right] \\ = \mathrm{E}\left[\sum_{i=1}^{i_{\max}} Y_i^2\right] \leq \mathrm{E}[7 + 4/n] = 7 + 4/n.\end{aligned}$$

So by Chebyshev's inequality,

$$\Pr\left[|U_{i_{\max}}| \geq 4\right] \leq \frac{7 + 4/n}{4^2} = 7/16 + 1/4n \leq 7/8,$$

when $n \geq 4$. But if $|U_{i_{\max}}| < 4$, we have $S_{\tau'} = S_\tau + U_{i_{\max}} \geq 13 - 4 = 9$. As the event $|U_{i_{\max}}| < 4$ occurs with conditional probability at least $1/8$, the total probability that $S_{\tau'} \geq 9$ is at least $(1/8)p_{13}$. ■

7.4 Full result

We are now ready to prove the main theorem of having a constant agreement parameter.

Theorem 7.10 *For sufficiently large n, Algorithm 7.1 implements a shared coin with constant agreement parameter.*

Proof: Let T be the total number of votes generated.

The total vote Z_i computed by any process in Line 16 is equal to S_T minus at most one vote for each process because of the termination bit. From Lemma 7.1, these unwritten votes have total size bounded by $1 + \sum_{i=1}^{T} X_i^2$. We show there is at least a constant probability that both $1 + \sum_{i=1}^{T} X_i^2 \leq 8 + 4/n$ and $S_T > 9$, which implies that for sufficiently large n there is a constant probability for having $Z_i > 0$ for all i, and therefore all processes agree on the value $+1$.

From Lemma 7.7, we have $\sum_{i=1}^{T} X_i^2 \leq 7 + 4/n$. From Lemma 7.9, the probability that $S_{\tau'} \leq 9$ is at most $1 - (1/8)p_{13}$. Therefore, for at least some constant δ, we have $S_T \geq S_{\tau'} > 9$ and $1 + \sum_{i=1}^{T} X_i^2 \leq 8 + 4/n$ with probability δ.

This proves that there is a constant probability of all processes deciding $+1$; the same results hold for -1 by symmetry. ■

Chapter 8

Randomized Set Agreement

In this chapter we present several algorithms for solving set-agreement with different parameters. In Section 8.1 we present a framework for randomized algorithms which solve $(k, k+1, n)$-agreement using a *multi-sided shared-coin* algorithm. We now formally define such a procedure, which is a generalization of a shared coin (which in our terms is a 2-sided shared coin). A $(k+1)$-*sided shared-coin* algorithm with *agreement parameter* δ is an algorithm in which every non-faulty process p produces an output value in $\{0, \ldots, k\}$, such that for every subset of size k there is probability at least δ that all the outputs are within that subset. Alternatively, for every value v in $\{0, \ldots, k\}$ there is probability at least δ that v is *not* the output of any process. We emphasize that unlike the requirement of set agreement, the probability of disagreement in a shared coin may be greater than 0. Notice that there are no inputs to this procedure.

In Section 8.3, we present set-agreement algorithms that are designed for agreeing on ℓ values out of $k+1$, for $\ell < k$. In particular, they can be used for the case $\ell = 1$, where the processes agree on the same value, i.e., for *multi-valued consensus*. By definition, solving multi-valued consensus is at least as hard as solving *binary consensus* (where the inputs are in the set $\{0, 1\}$, i.e., $k = 1$), and potentially harder. One algorithm uses multi-sided shared coins, while the other two embed binary consensus algorithms in various ways.

To the best of our knowledge, these are the first wait-free algorithms for set agreement in the shared-memory model under a strong adversary, other than binary consensus. Table 8.1 shows the properties of the different algorithms we present. For $\ell < k$ one of our algorithms is better than the others; however, intrigued by the question of whether multi-valued consensus is inherently harder

Algorithm	Parameters	Method	Individual Step Complexity	Total Step Complexity
Section 8.1	$k, k+1$	multi-sided shared coin	$O(n/k + k)$	$O(n^2/k + nk)$
Section 8.3.1	$\ell, k+1$	space reduction	$O(n(\log k - \log \ell))$	$O(n^2(\log k - \log \ell))$
Section 8.3.2	$\ell, k+1$	iterative	$O((k-\ell+1)k$ $+n(\log k - \log \ell))$	$O((k-\ell+1)nk$ $+n^2(\log k - \log \ell))$
Section 8.3.3	$1, k+1$	bit-by-bit	$O(n \log k)$	$O(n^2 \log k)$

Table 8.1: The set agreement algorithms presented in Chapter 8.

than binary consensus, we find the different methods interesting in hope that one of them could lead to a lower bound.

Finally, we note that in this chapter we will consider the set of processes as $\{p_0, \ldots, p_{n-1}\}$.

8.1 A $(k, k+1, n)$-Agreement Algorithm using a $(k+1)$-Sided Shared Coin

In this section we present a framework for randomized $(k, k+1, n)$-agreement algorithms. It is a generalization of the framework of Aspnes and Herlihy [12] for deriving a randomized binary consensus algorithm from a shared coin, and specifically follows the presentation given by Saks, Shavit, and Woll [66]. However, its complexity is improved by using multi-writer registers, based on the construction of Cheung [34].

We assume a $(k+1)$-sided shared-coin algorithm called $\mathsf{sharedCoin}_{k+1}$, with an agreement parameter δ_{k+1}. The set-agreement algorithm is given in Algorithm 8.1. Throughout this chapter, we assume that shared arrays are initialized to a special symbol $\perp$. Informally, the set-agreement algorithm proceeds by (asynchronous) phases, in which each process p writes its own preference to a shared array $Propose$, checks if the preferences agree on k values, and notes this in another shared array $Check$. If p indeed sees agreement, it also notes its preference in $Check$.

Process p then checks the agreement array $Check$. If p does not observe a note of disagreement, it decides on the value of its preference. Otherwise, if there is a note of disagreement, but also a

Algorithm 8.1 A $(k, k+1, n)$-agreement algorithm, code for p_i

Local variables: $r = 1$, $decide =$ false, $myValue = input$,
$myPropose = [\,]$, $myCheck = [\,]$
Shared arrays: $Propose[\,][0..k]$, $Check[\,]$[agree, disagree]

```
1:  while decide == false
2:      Propose[r][myValue] = true
3:      myPropose = collect(Propose[r])
4:      if the number of values in myPropose is at most k
5:          Check[r][agree] = ⟨true, myValue⟩
6:      else
7:          Check[r][disagree] = true
8:      myCheck = collect(Check[r])
9:      if myCheck[disagree] == false
10:         decide = true
11:     else if myCheck[agree] == ⟨true, v⟩
12:         myValue = v
13:     else if myCheck[agree] == false
14:         myValue = sharedCoin_{k+1}[r]
15:     r = r + 1
16: end while
17: return myValue
```

note of agreement, p adopts the value associated with the agreement notification as preference for the next phase. Finally, if there is only a notification of disagreement, the process participates in a $(k+1)$-sided shared-coin algorithm and prefers the output of the shared coin.

Lemma 8.1 *Consider a phase $r \geq 1$ and a non-faulty process p that finishes phase r. If all the processes that start phase r before p finishes it have at most k preferences in $\{v_1, \ldots, v_k\}$, then p decides $v \in \{v_1, \ldots, v_k\}$ in this phase r.*

Proof: We claim that p reads $Check[r]$[disagree] $==$ false in line 9 of phase r, and therefore decides in phase r. This will also imply that its decision value v is in $\{v_1, \ldots, v_k\}$, otherwise

p is among the processes that start phase r before p finishes, but does not have a preference in $\{v_1, \ldots, v_k\}$, which contradicts our assumption. Assume towards a contradiction, that p reads $Check[r]$[disagree] == true in line 9 of phase r. This implies that there is a process q that writes $Check[r]$[disagree] = true in line 7 of phase r, and this happens before p finishes. Therefore, q reads more than k values in $Propose[r]$ in line 3 of phase r, which means that there are $k+1$ processes that write $k+1$ different values to $Propose[r]$ in line 2 of phase r, and all this happens before p finishes. But this contradicts our assumption that all the processes that start phase r before a non-faulty process p finishes it have at most k preferences. ■

Lemma 8.1 implies validity, by applying it for phase $r = 1$. The next two lemmas are used to prove the agreement condition. Below, we use the notation $\langle$true, ?$\rangle$ for an entry in the array $Check$ which has *true* as its first element, and any value as its second element.

Lemma 8.2 *For every phase $r \geq 1$, all the processes that read $Check[r]$[agree] == $\langle$true, ?$\rangle$ and finish phase r have at most k different preferences at the end of phase r.*

Proof: We first claim that all the processes that write to $Check[r]$[agree] wrote at most k different preferences to $Propose[r]$. Assume, towards a contradiction, that among the processes that write to $Check[r]$[agree] there are $k+1$ processes $\{p_{i_1}, \ldots, p_{i_{k+1}}\}$ that wrote $k+1$ different preferences to $Propose[r]$. Let p_{i_j} be the last process to write to $Propose[r]$. When p_{i_j} collects $Propose[r]$ in line 3, it reads $k+1$ values, and therefore does not write to $Check[r]$[agree], which is a contradiction.

The above claim implies that at most k different preferences may be written to $Check[r]$[agree]. Since a process that reads $Check[r]$[agree] == $\langle$true, $v\rangle$ adopts v as its preference, at most k values can be a preference of such processes at the end of phase r. ■

Lemma 8.3 *For every phase $r \geq 1$, if processes decide on values in $\{v_1, \ldots, v_k\}$ in phase r, then every non-faulty process decides on a value in $\{v_1, \ldots, v_k\}$ in phase r', where r' is either r or $r+1$.*

Proof: We first claim that if a process decides v in phase r, then every non-faulty process that finishes phase r reads $Check[r]$[agree] == $\langle$true, ?$\rangle$. To prove the claim, let p be a process that decides v in phase r. Let q be a non-faulty process that finishes phase r, and assume towards a contradiction that q reads $Check[r]$[agree] == false. This implies that q collects $Check[r]$ in

line 8 before p writes to $Check[r]$ in line 5, and therefore p collects $Check[r]$ after q writes to $Check[r]$[disagree], which implies that p does not decide in phase r, a contradiction.

Now, let p be a process that decides in phase r, and let q be a non-faulty process. By the above claim, q reads $Check[r]$[agree] $== \langle$true, ?$\rangle$ in line 8. By Lemma 8.2, there are at most k different values that can become a preference of a process at the end of phase r. Therefore, if q decides at the end of phase r then it decides a value in $\{v_1, \ldots, v_k\}$. Otherwise, all the non-faulty processes write at most k preferences to $Propose[r+1]$, and by Lemma 8.1, they decide on one of these values at the end of phase $r+1$. ■

Lemma 8.3 implies agreement. Notice that both validity and agreement are *always* satisfied, and not only with probability 1. For termination, we prove the following lemma. Below, we denote the agreement parameter of the $(k+1)$-sided shared coin by $\delta = \delta_{k+1}$.

Lemma 8.4 *The expected number of phases until all non-faulty processes decide is at most* $1+1/\delta$.

Proof: For every subset $\{v_1, \ldots, v_k\} \subseteq \{0, \ldots, k\}$ there is a probability of at least δ for all processes that run sharedCoin$_{k+1}$ to output values in $\{v_1, \ldots, v_k\}$. Therefore, for any value v in $\{0, \ldots, k\}$, there is a probability of at least δ that v is not the output of any process running sharedCoin$_{k+1}$. This is because $\{0, \ldots, k\} \setminus \{v\}$ has probability of at least δ for containing the outputs of all the processes.

Consider a phase $r \geq 2$. By Lemma 8.2, all the processes that finish phase $r-1$ and in line 8 read $Check[r-1]$[agree] $== \langle$true, ?$\rangle$ propose at most k values to $Propose[r]$. The other processes propose to $Propose[r]$ a value obtained from their shared coin. Therefore, there is a probability of at least δ that all processes write at most k different values to $Propose[r]$, and by Lemma 8.1, decide by the end of phase r.

Therefore, after phase $r = 1$, the expected number of phases until all non-faulty processes decide, is the expectation of a geometrically distributed random variable with success probability at least δ, which is at most $1/\delta$.

For the first phase $r = 1$, the values written to $Propose[1]$ are the inputs and are therefore controlled by the adversary. This implies that the expected number of phases until all non-faulty processes decide is at most $1 + 1/\delta$. ■

Algorithm 8.2 A $(k+1)$-sided shared coin algorithm, code for process p_i

Local variables: $j = \lfloor \frac{ik}{n} \rfloor$

1: return $\mathsf{sharedCoin}[j] + j$

Consider a $(k+1)$-sided shared coin algorithm with an agreement parameter $\delta = \delta_{k+1}$, a total step complexity of $T = T_{k+1}$, and an individual step complexity of $I = I_{k+1}$. In each phase, a process takes $O(k)$ steps in addition to the I steps it takes in the $\mathsf{sharedCoin}_{k+1}$ algorithm. Combining this with Lemma 8.4, which bounds the expected number of phases until all non-faulty processes decide, gives:

Theorem 8.5 *Algorithm 8.1 solves $(k, k+1, n)$-agreement with $O(\frac{I+k}{\delta})$ individual step complexity and $O(\frac{T+nk}{\delta})$ total step complexity.*

8.2 A $(k+1)$-Sided Shared Coin

We present, in Algorithm 8.2, a $(k+1)$-sided shared-coin algorithm which is constructed by using k instances of a 2-sided shared coin. We statically partition the processes into k sets of at most $\frac{n}{k}$ processes each. That is, for every j, $0 \leq j \leq k-1$, we have a set $P_j = \left\{ p_{\frac{jn}{k}}, \ldots, p_{\frac{(j+1)n}{k}-1} \right\}$ (for $j = k-1$ the set may be smaller). The processes of each set P_j run a 2-sided shared-coin algorithm $\mathsf{sharedCoin}[j]$ and output the result plus the value j. The idea is that in order to have a value j that is not the output of any process, it is enough that all processes running $\mathsf{sharedCoin}[j-1]$ agree on the value 0 and therefore output $j-1$, and that all the processes running $\mathsf{sharedCoin}[j]$ agree on the value 1 and therefore output $j+1$.

Let $\delta = \delta_2$ be the agreement parameter of the 2-sided shared coin. We bound the agreement parameter of the $k+1$-sided shared coin in the next lemma.

Lemma 8.6 *Algorithm 8.2 is a $(k+1)$-sided shared coin with an agreement parameter δ^2.*

Proof: There is a probability of at least δ for all processes who run $\mathsf{sharedCoin}[j]$ to return the value j, and a probability of at least δ for all processes who run $\mathsf{sharedCoin}[j]$ to return the value $j+1$. Therefore, for any value in $\{0, \ldots, k\}$, there is a probability of at least δ^2 that this value is not the output of any process running $\mathsf{sharedCoin}[j]$, for any $0 \leq j \leq k-1$ (because $j = 0$ may

be the output only of sharedCoin$[0]$, $j = k$ only of sharedCoin$[k-1]$, and $j \in \{1, \ldots, k-1\}$ only of sharedCoin$[j-1]$ and sharedCoin$[j]$). Therefore, Algorithm 8.2 is a $(k+1)$-sided shared coin with an agreement parameter δ^2. ■

The next lemma gives the complexity of the $k+1$-sided shared coin, and follows immediately from the fact that each process runs a 2-sided shared coin algorithm for $\frac{n}{k}$ processes. Since the complexities depend on the number of processes t that may run an algorithm, we now carefully consider this in the notation. Let $I(t) = I_2(t)$ and $T(t) = T_2(t)$ be the individual and total step complexities, respectively, of the 2-sided shared coin with t processes.

Lemma 8.7 *Algorithm 8.2 has individual and total step complexities of $O(I(\frac{n}{k}))$ and $O(k \cdot T(\frac{n}{k}))$, respectively.*

Plugging Lemmas 8.6 and 8.7 into Theorem 8.5 gives:

Theorem 8.8 *Algorithm 8.1 solves $(k, k+1, n)$-agreement with individual step complexity of $O((I(\frac{n}{k}) + k)/\delta^2)$ and total step complexity of $O((k \cdot T(\frac{n}{k}) + nk)/\delta^2)$.*

By using an optimal 2-sided shared coin [11] with a constant agreement parameter, an individual step complexity of $O(t)$, and a total step complexity of $O(t^2)$, we get that Algorithm 8.2 is a $(k+1)$-sided shared coin with a constant agreement parameter, and individual and total step complexities of $O(\frac{n}{k})$ and $O(\frac{n^2}{k})$, respectively. Therefore, Algorithm 8.1 solves $(k, k+1, n)$-agreement with individual step complexity of $O(\frac{n}{k} + k)$ and total step complexity of $O(\frac{n^2}{k} + nk)$. Note that for $n \geq k^2$, Algorithm 8.1 has $O(\frac{n}{k})$ individual step complexity, and $O(\frac{n^2}{k})$ total step complexity, which are the same as the complexities of binary consensus divided by k.

8.3 $(\ell, k+1, n)$-Agreement Algorithms

In this section we construct several algorithms for the solving $(\ell, k+1, n)$-agreement, where $\ell < k$.

8.3.1 An $(\ell, k+1, n)$-Agreement Algorithm by Space Reduction

For agreeing on one value out of $\{0, \ldots, k\}$ we can get a total step complexity of $O(n^2 \log k)$ by reducing the possible values by half until we have one value. We later show how this construction can be used for agreeing on $\ell > 1$ values.

Algorithm 8.3 A $(1, k+1, n)$-agreement algorithm by space reduction, code for p_i

Local variables $myValue = input$, $myPair$, $mySide$

Shared arrays: $Agree[1..\lceil \log(k+1) \rceil][1..k/2^j]$,

$Values[1..\lceil \log(k+1) \rceil][1..k/2^j][0..1]$

1: for $j = 1 \ldots \lceil \log(k+1) \rceil$

2: $myPair = \lfloor \frac{myValue}{2^j} \rfloor$

3: if $myValue - myPair \cdot 2^j < 2^{j-1}$

4: $mySide = 0$

5: else $mySide = 1$

6: $Values[j][myPair][mySide] = myValue$

7: $side = Agree[j][myPair](mySide)$

8: $myValue = Values[j][myPair][side]$

9: end for

10: return $myValue$

In Algorithm 8.3 we assume an array $Agree$ of binary consensus instances, which a process can execute with a proposed value. Algorithm 8.3 can be modelled as a binary tree, where the processes begin at the leaves, which represent all of the values, and in every iteration j the processes agree on the value of the next node, going up to the root. This means that at most half of the suggested values are decided in each iteration. In addition, all decided values are valid because this is true for each node.

Lemma 8.9 *(Validity) For every j, the variable $myValue$ of a process p at the end of iteration j is an input of some process.*

Proof: The proof is by induction on j. The base case $j = 1$ is clear since $myValue$ is initialized with the input of the process. For the induction step, assume the lemma holds up to $j-1$, and notice that $myValue$ is updated only in line 8, to the value written in the $Values$ array in the location $side$ which is returned from the binary consensus algorithm. Since the consensus algorithm satisfies validity, $side$ has to be the input of some process to the consensus algorithm, and this only happens if that process first writes to that location in the $Values$ array in line 6. By the induction hypothesis, that value is the input of some process. ■

Lemma 8.10 *(Agreement) Every two process executing Algorithm 8.3 output the same value.*

Proof: We claim that there can be at most one value written to $Values[j][pair][side]$, and prove this by induction, where the base case is trivial since at the beginning a process writes to $Values[1][pair][side]$ only if its input value is $2 \cdot pair + side$. Assume this holds up to iteration $j-1$. By the agreement property of the consensus algorithm, all processes that execute $Agree[j-1][pair]$ output the same value. Therefore, in iteration j, only one value out of $\{2^j \cdot pair, \ldots, 2^j(pair+1)-1\}$ can be written to $Values[j][pair][side]$. The lemma follows by applying the claim to the root, which satisfies agreement. ■

Termination follows from the termination property of the binary consensus instances. For each j, a process executes one consensus algorithm, plus $O(1)$ additional accesses to shared variables. By using an optimal binary consensus algorithm where a process completes within $O(n)$ steps, this implies:

Theorem 8.11 *Algorithm 8.3 solves $(1, k+1, n)$-agreement with an individual step complexity of $O(n \log k)$ and a total step complexity of $O(n^2 \log k)$.*

Note that we can backstop this construction at any level j at the tree to get an agreement on $\ell = 2^{\log k - j}$ values. This means that instead of having j iterate from 1 to $\lceil \log(k+1) \rceil$, the algorithm changes so that j iterates from 1 to $\lceil \log(k+1) \rceil - \lceil \log \ell \rceil$. The individual step complexity is $O(n(\log k - \log \ell))$, and the total step complexity is $O(n^2(\log k - \log \ell))$.

8.3.2 An Iterative $(\ell, k+1, n)$-Agreement Algorithm

In Algorithm 8.4, we construct an $(\ell, k+1, n)$-agreement algorithm by iterating Algorithm 8.1 and reducing the number of possible values by one until all processes output no more than ℓ values. The idea is that the processes execute consecutive iterations of $(s, s+1, n)$-agreement algorithms for values of s decreasing from k to ℓ. In each iteration the number of possible values is reduced until it reaches the desired bound ℓ.

This procedure is less trivial than it may appear because, for example, after the first iteration outputs no more than k values out of $k+1$, in order to decide on $k-1$ out of the k values that are possible, the processes need to know *which* are the k possible values. However, careful inspection

shows that they need to know these k values only if they disagree upon choosing the $k-1$ values out of them. In this case, a process that sees k values indeed knows which are these values among the initial $k+1$.

We now present the pseudocode of Algorithm 8.4 which solves $(\ell, k+1, n)$-agreement by iteratively decreasing the number of possible values using Algorithm 8.1, as discussed in Section 8.3.2.

Notice that Algorithm 8.1 is correct for agreeing on k values out of $k+1$ values, even if the $k+1$ possible input values are not necessarily $\{0, \ldots, k\}$, as long as they are a fixed and known set $\{v_0, \ldots, v_k\}$. This is done by having a bijective mapping between the two sets.

The following lemma guarantees the correctness of the algorithm.

Lemma 8.12 *For each iteration s, $\ell \leq s \leq k$, the number of different values that appear in the $myValue$ variables of the processes that finish iteration s is at most s, and each of these values is the input of some process.*

Proof: The proof is by induction over the iterations, where the base case is for $s = k$ and its proof is identical to that of Algorithm 8.1. For the induction step, we assume the lemma holds up to $s+1$ and prove it for s. A process finishes iteration s when it assigns $decide =$ true in line 13. This can only happen after it reads $myCheck[\text{disagree}] ==$ false in line 10, which implies that the number of different entries in $myPropose$ that contain $true$ is at most s. Moreover, every value that is written to the $Propose[s]$ array is the $myValue$ variable of some process at the end of iteration $s+1$, and therefore is the input of some process, by the induction hypothesis. ■

Applying Lemma 8.12 to $s = \ell$ gives the validity and agreement properties. This leads to the following theorem:

Theorem 8.13 *Algorithm 8.4 solves $(\ell, k+1, n)$-agreement with $O(\sum_{s=k}^{\ell} \frac{I_{s+1}+k}{\delta_{s+1}})$ individual step complexity and $O(\sum_{s=k}^{\ell} \frac{T_{s+1}+nk}{\delta_{s+1}})$ total step complexity, where δ_{s+1}, I_{s+1}, and T_{s+1} are the agreement parameter, individual step complexity, and total step complexity, respectively, of the $(s+1)$-sided shared coins.*

Proof: For each value of s, a process runs an iteration of the agreement algorithm for s out of $s+1$ values. By an analog of Theorem 8.1, this takes $O(\frac{I_{s+1}+k}{\delta_{s+1}})$ individual step complexity, and $O(\frac{T_{s+1}+nk}{\delta_{s+1}})$ individual step complexity. Notice that we add $O(k)$ steps for collecting the arrays and

Algorithm 8.4 An $(\ell, k+1, n)$-agreement algorithm, code for process p_i

local variables: $myValue$, $myPropose = [0..k]$,
$myCheck = [\text{agree}, \text{disagree}]$,$s$,$m$,$r$,$decide$
shared arrays: $Propose[1..k][\,][0..k]$,
$Check[1..k][\,][\text{agree}, \text{disagree}]$

1: for $s = k$ down to ℓ
2: $r = 1$
3: $decide = \text{false}$
4: while $decide == \text{false}$
5: $Propose[s][r][myValue] = \text{true}$
6: $myPropose = \text{collect}(Propose[s][r])$
7: if the number of entries in $myPropose$ that contains true
is at most s
8: $Check[s][r][\text{agree}] = \langle \text{true}, myValue \rangle$
9: else
10: $Check[s][r][\text{disagree}] = \text{true}$
11: $myCheck = \text{collect}(Check[s][r])$
12: if $myCheck[\text{disagree}] == \text{false}$
13: $decide = \text{true}$
14: else if $myCheck[\text{agree}] == \langle \text{true}, v \rangle$
15: $myValue = v$
16: else if $myCheck[\text{agree}] == \text{false}$
17: $m = \mathsf{sharedCoin}_{s+1}[r]$
18: $myValue =$ the m-th entry in $myPropose$ that
contains true // At most $s+1$ such values
19: $r = r + 1$
20: end while
21: end for
22: return $myValue$

not $O(s)$ steps, since it may be that a process does not know which are the $s+1$ current possible values among the initial $k+1$ values.

Summing over all iterations gives the resulting complexities. ■

When using the $(s+1)$-sided shared coins of Section 8.2 we have:

Theorem 8.14 *Algorithm 8.4 solves $(\ell, k+1, n)$-agreement with $O((k-\ell+1)k+n(\log k-\log \ell))$ individual step complexity and $O((k-\ell+1)nk+n^2(\log k-\log \ell))$ total step complexity.*

Proof: For the individual step complexity we have:

$$\begin{aligned}\sum_{s=k}^{\ell} \frac{I_{s+1}+k}{\delta_{s+1}} &= O(\sum_{s=k}^{\ell} \frac{n}{s}+k) \\ &= O((k-\ell+1)k+n\sum_{s=k}^{\ell}\frac{1}{s}) \\ &= O((k-\ell+1)k+n(\log k-\log \ell)),\end{aligned}$$

where the last equality follows from the fact that the harmonic series $H_k=\sum_{s=1}^{k}\frac{1}{s}$ is in the order of $\log k$. Similarly, we have that the total step complexity is $O((k-\ell+1)nk+n^2(\log k-\log \ell))$.

■

Note that for $\ell=1$, i.e., for agreeing on exactly one value out of the initial $k+1$ possible inputs, we get an individual step complexity of $O((k-\ell+1)k+n(\log k-\log \ell))=O(k^2+n\log k)$, and a total step complexity of $O((k-\ell+1)nk+n^2(\log k-\log \ell))=O(nk^2+n^2\log k)$.

8.3.3 A Bit-by-Bit $(1, k+1, n)$-Agreement Algorithm

For agreeing on one value out of $\{0,\ldots,k\}$ we construct Algorithm 8.5, which agrees on each bit at a time while making sure that the final value is valid. A similar construction appears in [72, Chapter 9], but does not address the validity condition. In this algorithm, obtaining validity is a crucial point in the construction, since simply agreeing on enough bits does not guarantee an output that is the input of some process.

The idea of our algorithm is that in every iteration j, all the $myValue$ local variables share the same first $j-1$ bits, and they are all valid values (each is the input of at least one process).

Algorithm 8.5 A $(1, k+1, n)$-agreement algorithm by agreeing on $\log k$ bits, code for p_i

local variables: $myValue = input$, $myPropose = [0..\log k]$,
$myCheck = [\text{agree}, \text{disagree}]$,$r = 0$,$decide = \text{false}$
shared arrays: $Propose[1..k][\,][0..\log k]$,
$Check[1..k][\,][\text{agree}, \text{disagree}]$

1: for $j = 1 \ldots \lceil \log(k+1) \rceil$
2: while ($decide == \text{false}$)
3: $r+ = 1$
4: $Propose[j][r][myValue[j]] = myValue$
5: $myPropose = \text{collect}(Propose[j][r])$
6: if $myPropose[0] \neq \perp$ and $myPropose[1] \neq \perp$
7: $Check[j][r][\text{disagree}] = myPropose$
8: else $Check[j][r][\text{agree}] = myValue$
9: $myCheck = \text{collect}(Check[j][r])$
10: if $myCheck[\text{disagree}] \neq \perp$
11: $coin = \text{sharedCoin}_2(j, r)$
12: if $myCheck[\text{agree}] \neq \perp$
13: $myValue = Propose[j][r][myCheck[\text{agree}]]$
14: else $myValue = myCheck[\text{disagree}][coin]$
15: else $decide = true$ and $r = 0$
16: end while
17: end for
18: return $myValue$

We now present the pseudocode of Algorithm 8.5 which solves $(1, k+1, n)$-agreement by agreeing on every bit of the value, as discussed in Section 8.3.3.

Lemma 8.15 *For every j, $1 \leq j \leq \lceil \log(k+1) \rceil$, at the beginning of iteration j every process has $myValue$ that is the input of some process, and all the processes have $myValue$ with the same first $j-1$ bits.*

Proof: The proof is by induction on j. The base case for $j = 1$ clearly holds since at the beginning of the algorithm $myValue$ is initialized to the input of the process, and $j - 1 = 0$ so there is no requirement from the first bits of $myValue$.

Induction step: Assume that the lemma holds up to value $j-1$. That is, the variable $myValue$ of all processes at the beginning of iteration $j-1$ has the same $j-2$ first bits, and they are all inputs of processes.

First, we notice that in iteration $j-1$ the variable $myValue$ can only change to a value written in the $Propose$ array in line 13, or to a value written in the $Check$ array in line 14. This implies that $myValue$ is always an input of some process.

Next, assume that at the end of the iteration processes p and q have $myValue$ variables with different first $j-1$ bits. By the induction hypothesis, this implies that their $j-1$-th bit is different. Let r be the first phase in which such two processes exist and decide in that phase. Assume, without loss of generality, that p executes line 4 after q does. This implies that when p reads the array $Propose$ in line 5, both entries are non-empty. But then p writes its value into the $disagree$ location of the array $Check$ and therefore cannot decide in that phase. ■

Lemma 8.15, in an analog to Section 8.1, implies validity and agreement.

We denote by $\delta = \delta_2$ the agreement parameter of the 2-sided shared coin, and $T = T_2$ and $I = I_2$ are its total and individual step complexities, respectively.

Theorem 8.16 *Algorithm 8.5 solves $(1, k+1, n)$-agreement with $O(\lceil \log(k+1) \rceil \cdot \frac{I}{\delta})$ individual step complexity and $O(\lceil \log(k+1) \rceil \frac{T}{\delta})$ total step complexity.*

Proof: In each iteration j, $1 \leq j \leq \lceil \log(k+1) \rceil$, by an analog to Lemma 8.4, the expected number of phases until all non-faulty process decide is $1 + 1/\delta$ which is $O(\frac{1}{\delta})$. In each phase, a process takes $O(1)$ steps in addition to the I steps it takes in the sharedCoin$_2$ algorithm. Therefore, the

individual step complexity of Algorithm 8.5 is $O(\lceil \log (k+1) \rceil \cdot \frac{I}{\delta})$, and the total step complexity is $O(\lceil \log (k+1) \rceil \frac{T}{\delta})$. ■

Using an optimal shred coin with a constant agreement parameter, an individual step complexity of $O(n)$, and a total step complexity of $O(n^2)$, we get a $(1, k+1, n)$-agreement algorithm with an individual step complexity of $O(n \log k)$ and a total step complexity of $O(n^2 \log k)$. Notice that the step complexity could be improved if agreement on the bits could be run in parallel. However, this is not trivial because of the need to maintain validity.

Part II

Lower Bounds for Randomized Consensus

Chapter 9

Layering Randomized Executions

This part of the thesis presents lower bounds for randomized consensus under a weak adversary (Chapter 10) and under a strong adversary (Chapter 11). We begin, in this chapter, by providing formal definitions and by setting the common grounds to both lower bounds, namely, layered randomized executions. This includes a formal presentation of the two types of adversaries that captures the difference in their ability to control the execution.

9.1 Preliminaries

The lower bounds presented in this part also address the *message-passing* model, in addition to the shared-memory model. We therefore define a step of a process as consisting of some local computation, including an arbitrary number of local coin flips and one communication operation, which depends on the communication model.

In a message passing system, processes communicate by sending and receiving messages: the communication operation of a process is sending messages to some subset of the processes, and receiving messages from some subset of them. For the lower bounds, we assume that a process sends a message to all the processes in each step. In a shared memory system processes communicate by reading and writing to shared (atomic) registers as defined in Section 3; each step of a process is either a read or a write to some register. The types of the registers that are assumed will be explicitly defined later.

For the purpose of the lower bound, we restrict our attention to a constrained set of executions,

which proceed in layers. An f-*layer* is a sequence of at least $n - f$ distinct process id's. When executing a layer L, each process $p \in L$ takes a step, in the order specified by the layer.

An f-*execution* is an execution of the algorithm under a (finite or infinite) sequence of f-layers.

A configuration C consists of the local states of all the processes, and the values of all the registers. We will consider only configurations that are reachable by finite sequence of f-layers.

We define faulty processes as follows: a process p_i is *non-faulty* in layer r if it appears in the layer. A process p_i *crashes* in layer r if it does not take a step in any layer $\ell \geq r$. A process is *skipped* in layer r, if it does not appear in layer r but appears in one of the following layers.

Since we consider randomized algorithms, for each configuration C there is a fixed probability for every step a process will perform when next scheduled. Denote by X_i^C the probability space of the steps that process p_i will preform, if scheduled by the adversary. The probability space X_i^C depends only on the local state of p_i in configuration C, and therefore, delaying p_i does not change this probability space.

Let $X^C = X_1^C \times X_2^C \times \cdots \times X_n^C$ be the product probability space. A vector $\vec{y} \in X^C$ represents a possible result of the local coin flips from a configuration C.

9.2 Adversaries

Since we are discussing randomized algorithms, different assumptions on the power of the adversary may yield different results. We now model two types of adversaries, one called *strong* and the other *weak*. We first define a strong adversary, followed by the definition of a weak adversary as a restricted case. However, the lower bounds are presented in reverse order, since the case of the strong adversary is more involved.

9.2.1 Strong Adversary

A *strong* adversary observes the processes' local coin flips, and chooses the next f-layer knowing what is the next step each process will take. The adversary applies a function σ to choose the next f-layer to execute for each configuration C and vector $\vec{y} \in X^C$, i.e.,

$$\sigma : \{(C, \vec{y}) \mid C \text{ is a configuration and } \vec{y} \in X^C\} \rightarrow \{L \mid L \text{ is an } f\text{-layer}\}.$$

When the configuration C is clear from the context we will use the abbreviation $\sigma(\vec{y}) = L_{\vec{y}}$.

Denote by $(C, \vec{y}, L_{\vec{y}})$ the configuration that is reached by applying steps of the processes in $L_{\vec{y}}$, for a specific vector $\vec{y} \in X^C$. Then $C \circ \sigma$ is a random variable whose values are the configurations $(C, \vec{y}, L_{\vec{y}})$, when $\vec{y}$ is drawn from the probability space X^C.

An *f-adversary* $\sigma = \sigma_1, \sigma_2, \ldots$ is a (finite or infinite) sequence of functions.

Given a configuration C and a finite prefix $\sigma_\ell = \sigma_1, \sigma_2, \ldots, \sigma_\ell$ of the adversary σ, $C \circ \sigma_\ell$ is a random variable whose values are the configurations that can be reached by the algorithm. For every vector $\vec{y}_1 \in X^C$, by abuse of notation, let $\Pr[\vec{y}_1] = \Pr[\vec{y}_1 \text{ is drawn from } X^C]$ denote the probability of $\vec{y}_1$ in the probability space X^C. The probability that a configuration C' is reached is defined inductively[1]:

$$\Pr[C \circ \sigma_\ell \text{ is } C'] = \sum_{\vec{y}_1 \in X^C} \Pr[\vec{y}_1] \cdot \Pr[(C, \vec{y}_1, L_{\vec{y}_1}) \circ \sigma'_\ell \text{ is } C'],$$

where σ'_ℓ is the remainder of the prefix after σ_1, i.e., $\sigma'_\ell = \sigma_2, \ldots \sigma_\ell$, and the basis of the induction for $\sigma_1 = \sigma_1$ is:

$$\Pr[C \circ \sigma_1 \text{ is } C'] = \sum_{\vec{y}_1 \in X^C} \Pr[\vec{y}_1] \cdot \ \chi_{C'}(\vec{y}_1),$$

where $\chi_{C'}(\vec{y}_1) = \chi_{C'}(C, \sigma_1, \vec{y}_1)$ characterizes whether the configuration C' is reached if $\vec{y}_1$ is drawn, i.e.,

$$\chi_{C'}(\vec{y}_1) = \begin{cases} 1 & (C, \vec{y}_1, L_{\vec{y}_1}) \text{ is } C' \\ 0 & \text{otherwise.} \end{cases}$$

The probability of deciding v when executing the algorithm under σ from the configuration C is defined as follows: if C is a configuration in which there is a decision v, then $\Pr[\text{decision from } C \text{ under } \sigma \text{ is } v] = 1$, if C is a configuration in which there is a decision $\bar{v}$, then $\Pr[\text{decision from } C \text{ under } \sigma \text{ is } v] = 0$, otherwise,

$$\Pr[\text{decision from } C \text{ under } \sigma \text{ is } v] = \sum_{\vec{y}_1 \in X^C} \Pr[\vec{y}_1] \cdot \Pr[\text{decision from } (C, \vec{y}_1, L_{\vec{y}_1}) \text{ under } \sigma' \text{ is } v],$$

where σ' is the remainder of the adversary after σ_1, i.e., $\sigma' = \sigma_2, \sigma_3, \ldots$.

[1]For simplicity, we assume that all the probability spaces are discrete, but a similar treatment holds for arbitrary probability spaces.

9.2.2 Weak Adversary

A *weak* adversary is non-adaptive and decides the entire schedule in advance. The adversary does not observe neither the results of any local coins a process flips, nor any operation a process performs. We model this adversary as applying the function σ based only on the number of layers that have been scheduled, i.e.,

$$\sigma : \mathbb{N} \to \{L \mid L \text{ is an } f\text{-layer}\}.$$

In other words, a weak f-adversary is a (finite or infinite) sequence of layers $\sigma = L_1, L_2, \ldots$.

An adversary σ, together with an initial configuration I and n coin-flip strings $\vec{c} = (c_1, \cdots, c_n)$, determine an *execution* $\alpha(\sigma, \vec{c}, I)$. For a finite adversary σ, we identify the execution $\alpha(\sigma, \vec{c}, I)$ with the configuration it results in.

9.3 Manipulating Layers

Like many impossibility results, our proof relies on having configurations that are indistinguishable to all processes, except some set P. Intuitively, two configurations C and C' are indistinguishable to a process p if it cannot tell the difference between them. The idea behind identifying indistinguishable configurations is that processes that do not distinguish between them do the same thing in any extension where they are the only processes taking steps. Specifically, they decide the upon same value in the same extension from C and C'. Thus, comparing what happens in indistinguishable executions allows us to reason about the decision value in different executions, which is one of the main techniques in deriving lower bounds in distributed computing.

The formal definition of indistinguishability is model-dependent.

In the message-passing model, we say that the configurations C and C' are indistinguishable to the set of processes P and denote $C \overset{P}{\sim} C'$, if each process in P goes through the same local states throughout both executions up to C and C', More specifically, in both executions each process in P sends and receives the same messages, in the same order. We further require that processes in P are not crashed up to C and C'.

In the shared-memory model, the definition of indistinguishability is slightly different than in the message-passing model. For two configurations C and C' to be indistinguishable to a process

p, we not only require p to have the same local states throughout both executions (which implies that in both executions p performs the same shared-memory operations, including reading the same values from registers), but also that the values of the shared registers are the same throughout both; otherwise, for example, having p perform a read operation after C and C' might result in different executions. However, we allow the value of a shared register to be different in C and C' if it is no longer accessed by any process. This slight modification still captures the requirement that if a process p decides v in C and does not distinguish between C and C' then it also decides in C' on the same value v. We say that the configurations C and C' are indistinguishable to the set of processes P and denote $C \overset{P}{\sim} C'$, if the state of all processes that are in P is equal in both configurations C and C', these processes are not crashed in C and C', and the values of all the registers are equal. Similarly, we denote $C \overset{\neg P}{\sim} C'$ if the state of all processes that are *not* in P is equal in both configurations C and C', and the values of all the registers are equal. In both communication models we write $C \overset{p}{\sim} C'$ or $C \overset{\neg p}{\sim} C'$ when $P = \{p\}$.

In order to obtain indistinguishable configurations, we manipulate schedules by performing very small changes to the order of processes in a given layer. Every small change results in configurations that are indistinguishable to some processes. Intuitively, after many small changes we get a *chain* of indistinguishable configurations, which we formally define later. These chains allow us to argue about the decision value in each execution, and derive our lower bounds.

In the remainder of this section, we show a couple of manipulations that can be done to layers in the shared memory model and result in indistinguishable configurations (additional manipulations of layers appear separately in Chapters 10 and 11). We first consider a shared-memory model where processes communicate through multi-writer registers, and use a simplifying assumption that each read step accesses the registers of all processes. We call this the *multi-writer cheap-snapshot* model, since each register is written to by any process, and all registers are read by any process in a single snapshot. This snapshot is charged one step, hence the term "cheap".

We manipulate sets of processes, and then consider them as singletons when manipulations on single processes are needed. We consider a layer L as a sequence of disjoint sets of processes $L = [P_{i_1} \ldots P_{i_\ell}]$, where for every j, $1 \leq j \leq \ell$, all the processes in P_{i_j} perform the same operation: either a write operation or a cheap-snapshot operation.

The following claim handles swapping the order of consecutive sets of processes in a layer.

Claim 9.1 *Let C be a configuration, let $L = [P_{i_1}, \ldots, P_{i_\ell}]$ be a layer, and let $L' = [P_{i_1}, \ldots, P_{i_{j-1}}, P_{i_{j+1}}, \ldots, P_{i_\ell}]$ be the layer L after swapping P_{i_j} and $P_{i_{j+1}}$. If processes in P_{i_j} and $P_{i_{j+1}}$ do not write to the same registers, then $(C, \vec{y}_1, L) \overset{\neg P_{i_j}}{\sim} (C, \vec{y}_1, L')$ or $(C, \vec{y}_1, L) \overset{\neg P_{i_{j+1}}}{\sim} (C, \vec{y}_1, L')$.*

Proof: If all processes in P_{i_j} and $P_{i_{j+1}}$ access different registers or all of them perform a cheap-snapshot operation, then $(C, \vec{y}_1, L) \overset{\{p_1,\ldots,p_n\}}{\sim} (C, \vec{y}_1, L')$. If processes in P_{i_j} perform a cheap snapshot operation and processes in $P_{i_{j+1}}$ write, then $(C, \vec{y}_1, L) \overset{\neg P_{i_j}}{\sim} (C, \vec{y}_1, L')$, and otherwise $(C, \vec{y}_1, L) \overset{\neg P_{i_{j+1}}}{\sim} (C, \vec{y}_1, L')$. ■

The above claim assumes that the processes in the two swapped sets do not write to the same registers. The case where they do write to the same registers is more complex, since swapping such processes might change values of registers, and effect the rest of the execution. Instead of swapping two such sets of processes, we first remove the first set from the layer, attach it to the end of the modified layer, and finally swap it in reverse order until it reaches the desired location in the layer. The next claim address the first component of this manipulation, which removes such a set of processes from the layer.

Claim 9.2 *Let C be a configuration, let $L = [P_{i_1}, \ldots, P_{i_\ell}]$ be a layer where for some j, $1 \leq j < \ell$, all the processes in P_{i_j} and $P_{i_{j+1}}$ write to the same register R, and let $L' = [P_{i_1}, \ldots, P_{i_{j-1}}, P_{i_{j+1}}, \ldots, P_{i_\ell}]$ be the layer L after removing P_{i_j}. Then $(C, \vec{y}_1, L) \overset{\neg\{P_{i_j}\}}{\sim} (C, \vec{y}_1, L')$.*

Proof: Since all the processes in P_{i_j} and $P_{i_{j+1}}$ write to the same register R, after the processes in $P_{i_{j+1}}$ take steps in both layers the values of all the registers are same, and so are the local states of all the processes except those in P_{i_j}. This implies that this is also the case after executing the whole layers, and therefore $(C, \vec{y}_1, L) \overset{\neg P_{i_j}}{\sim} (C, \vec{y}_1, L')$. ■

The remaining component of attaching the set of processes to the end of the modified layer will be handled separately in Section 11.1, as it involves further definitions required for the case of a strong adversary.

Chapter 10

A Lower Bound for a Weak Adversary

In this chapter, we obtain our lower bounds for randomized consensus under a weak adversary in the different communication models. We begin by introducing the framework we use, which is common to all models.

As mentioned in Section 9.3, our lower bound under a weak adversary will make use of indistinguishable chains of executions. We proceed to formally define indistinguishability chains, as follows.

Recall that we identify an execution α with the configuration it results in. Given two executions α_1 and α_2 with the same n coin-flip strings $\vec{c} = (c_1, \cdots, c_n)$, we denote $\alpha_1 \overset{p_i}{\sim} \alpha_2$ if process p_i does not distinguish between α_1 and α_2, and does not crash in them. In this case, p_i decides on the same value in α_1 and in α_2. We denote $\alpha_1 \approx_m \alpha_2$ if there is a chain of executions $\beta_1, \cdots, \beta_{m+1}$ such that

$$\alpha_1 = \beta_1 \overset{p_{i_1}}{\sim} \beta_2 \cdots \overset{p_{i_m}}{\sim} \beta_{m+1} = \alpha_2 \ .$$

We call such a chain an *indistinguishability chain* of length $m+1$. Clearly, if $\alpha \approx_m \beta \approx_{m'} \gamma$ then $\alpha \approx_{m+m'} \gamma$, for every pair of integers m and m'. Moreover, notice that this relation is commutative, i.e., if $\alpha_1 \approx_m \alpha_2$ then $\alpha_2 \approx_m \alpha_1$.

For every pair of consecutive executions in the chain, there is a process that decides on the same value in both executions. By the agreement condition, the decision in α_1 and in α_2 must be the same. This is the main idea of the lower bound proof, which is captured in Theorem 10.1: we take two executions that must have different agreement values and construct an indistinguishability chain between them, which bounds the probability of terminating in terms of the length of the

chain. Two such executions exist by the validity condition, as we formalize next.

We partition the processes into $S = \max\{3, \lceil \frac{n}{f} \rceil\}$ sets $P_1, \ldots, P_S$, each with at most f processes. For example, if $n > 2f$, $P_i = \{p_{(i-1)f+1}, \cdots, p_{i \cdot f}\}$ for every i, $1 \leq i < S$, and $P_S = \{p_{(S-1)f+1}, \cdots, p_n\}$.

Consider initial configurations $C_0, \ldots, C_S$, such that in C_0 all the inputs are 0, and in C_i, $1 \leq i \leq S$, all processes in $P_1, \ldots, P_i$ have input 1 and all other processes have input 0; in particular, in C_S all processes have input 1.

Definition 10.1 *For a schedule σ, let $crash(\sigma, p, r)$ be the schedule that is the same as σ, except that p crashes in layer r, i.e., does not take a step in any layer $\ell \geq r$. For a set P of processes, $crash(\sigma, P, r)$ is defined similarly.*

Let σ_{full} be the full synchronous schedule with k layers, in which no process fails. The next theorem is the main tool for bounding q_k as a function of m, the length of an indistinguishability chain. This theorem distills the technique we borrow from [36]. At the end of Section 10.1 we discuss how asynchrony allows to construct shorter chains.

Theorem 10.1 *Assume there is an integer m such that for all sequences of coins $\vec{c}$, $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$. Then the probability that A does not terminate after $k(n-f)$ steps is $q_k \geq \frac{1}{m+1}$.*

Proof: Assume, by way of contradiction, that $q_k(m+1) < 1$. Since $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$, there is a chain of $m+1$ executions,

$$\alpha(\sigma_{full}, \vec{c}, C_0) = \beta_1 \overset{p_{i_1}}{\sim} \beta_2 \cdots \overset{p_{i_m}}{\sim} \beta_{m+1} = \alpha(\sigma_{full}, \vec{c}, C_S) \ .$$

(See Figure 10.1.) The probability that A does not terminate in at least one of these $m+1$ executions is at most $q_k(m+1)$. By assumption, $q_k(m+1) < 1$, and hence, the set B of sequences of coins $\vec{c}$ such that A terminates in all $m+1$ executions has probability $\Pr[\vec{c} \in B] > 0$. Since $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$, the agreement condition implies that the decision in all $m+1$ executions is the same. However, the validity condition implies that the decision in $\alpha(\sigma_{full}, \vec{c}, C_0)$ is 0, and the decision in $\alpha(\sigma_{full}, \vec{c}, C_S)$ is 1, which is a contradiction. ■

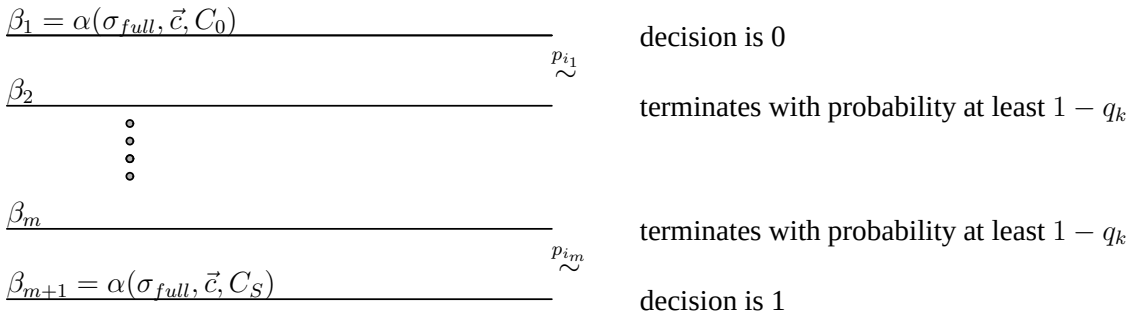


Figure 10.1: Illustration for the proof of Theorem 10.1.

A slight extension of the above theorem handles *Monte-Carlo* algorithms, where processes may terminate without agreement with some small probability ϵ. This extension is presented in Section 10.3.

The statement of Theorem 10.1 indicates that our goal is to show the existence of an integer m such that $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$; clearly, the smaller m, the higher the lower bound. The next lemma comes in handy when we construct these chains.

Lemma 10.2 *Assume there is an integer m such that for every schedule σ, initial configuration I, sequence of coins $\vec{c}$ and set P_i, $\alpha(\sigma, \vec{c}, I) \approx_m \alpha(crash(\sigma, P_i, 1), \vec{c}, I)$. Then $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_{S(2m+1)} \alpha(\sigma_{full}, \vec{c}, C_S)$, for every sequence of coins $\vec{c}$.*

Proof: Consider the schedules $\sigma_0 = \sigma_{full}$, and $\sigma_i = crash(\sigma_0, P_i, 1)$ for every i, $1 \leq i \leq S$, and the corresponding executions $\alpha_{i,j} = \alpha(\sigma_i, \vec{c}, C_j)$ for every i and j, $1 \leq i \leq S$ and $0 \leq j \leq S$. Note that the execution $\alpha_{i,j}$ starts from the initial configuration C_j with a schedule which is almost full, except that processes in P_i never take steps.

By assumption, $\alpha_{0,j} \approx_m \alpha_{i,j}$ for every i, $1 \leq i \leq S$, and every j, $0 \leq j \leq S$. (See Figure 10.2.) Since processes in P_i are crashed in σ_i for every i, $1 \leq i \leq S$, we have that $\alpha_{i,i-1} \overset{p}{\sim} \alpha_{i,i}$, for every process $p \in P \setminus P_i$. This implies that $\alpha_{i,i-1} \approx_1 \alpha_{i,i}$, for every i, $1 \leq i \leq S$. Thus,

$$\alpha(\sigma_{full}, \vec{c}, C_0) = \alpha_{0,0} \approx_m \alpha_{1,0} \approx_1 \alpha_{1,1} \approx_m \alpha_{0,1} \approx_m \alpha_{2,1} \approx_1 \alpha_{2,2} \cdots \alpha_{S,S} \approx_m \alpha_{0,S} = \alpha(\sigma_{full}, \vec{c}, C_S)\,.$$

Therefore, $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_{S(2m+1)} \alpha(\sigma_{full}, \vec{c}, C_S)$. ■

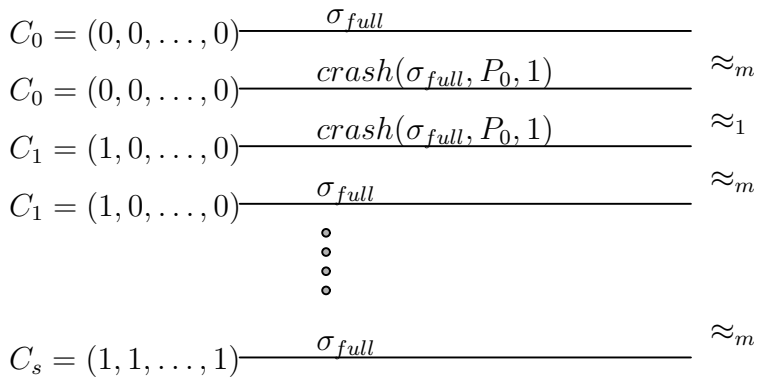


Figure 10.2: Illustration for the proof of Lemma 10.2.

10.1 Tradeoff for the Message-Passing Model

In this section we derive the lower bound for the message-passing model. Notice that in the message-passing model, since a step consists of both sending and receiving messages, a layer L is not only a sequence of processes, but also specifies for each process $p \in L$ the set of processes it receives a message from (recall that we assumed that it sends messages to all processes). The reception of messages in a certain layer is done after all messages of that layer are sent, and therefore the order of processes in a layer is insignificant.

Formally, an f-layer is a sequence $p_{i_1}, \ldots, p_{i_m}$ of distinct process id's, followed by a sequence $M_{i_1}, \ldots, M_{i_m}$ of subsets of process id's, where M_{i_j} is the set of process id's from which p_{i_j} receives a message in this layer. In the executions we construct, a message is either delivered in the same layer, or it is delayed and delivered after the last layer, and is effectively omitted in the execution.

Recall that the processes are partitioned into $S = \max\{3, \lceil \frac{n}{f} \rceil\}$ sets $P_1, \ldots, P_S$, each with at most f processes. We manipulate schedules in order to delay messages, as follows.

Definition 10.2 *Let σ be a finite schedule. Let $delay(\sigma, P_i, P_j, r)$ be the schedule that is the same as σ, except that the messages sent by processes in P_i in layer r are received by processes in P_j only after the last layer. More formally, if M_p is the subset of processes that a process p receives a message from in layer r in σ, then for every process $p \in P_j$ the subset of processes that it receives a message from in layer r in $delay(\sigma, P_i, P_j, r)$ is $M_p \setminus P_i$.*

Clearly, at the end of layer r, any process not in P_j does not distinguish between the execution

so far of a schedule σ and an execution so far of $delay(\sigma, P_i, P_j, r)$. Therefore we have:

Lemma 10.3 *Let σ be a schedule with k layers. For any sequences of coins $\vec{c}$, and initial configuration I, at the end of layer r only processes in P_j distinguish between $\alpha(\sigma, \vec{c}, I)$ and $\alpha(delay(\sigma, P_i, P_j, r), \vec{c}, I)$.*

Recall that $S = \max\{3, \lceil \frac{n}{f} \rceil\}$ is the number of sets P_i. We define the following recursive function for every r and k, $1 \leq r \leq k$:

$$m_{r,k} = \begin{cases} S & \text{if } r = k \\ (2(S-1)+1)m_{r+1,k} + S & \text{if } 1 \leq r < k \end{cases}$$

A simple induction shows that $m_{r,k} \leq (2S)^{k-r+1}$.

The following lemma proves that $m_{1,k}$ is the integer required in Lemma 10.2 for the message-passing model, by inductively constructing indistinguishability chains between executions in which a set of processes may crash from a certain layer r.

Lemma 10.4 *Let σ be a schedule with k layers such that for some r, $1 \leq r \leq k$, no process is skipped in layers $r, r+1, \ldots, k$. Then $\alpha(\sigma, \vec{c}, I) \approx_{m_{r,k}} \alpha(crash(\sigma, P_i, r), \vec{c}, I)$ for every sequence of coins $\vec{c}$, every initial configuration I, and every $i \in \{1, \ldots, S\}$.*

Proof: Let $\sigma = \sigma_0$. Throughout the proof we denote $\alpha_i = \alpha(\sigma_i, \vec{c}, I)$ for any schedule σ_i. The proof is by backwards induction on r.

Base case: $r = k$. We construct the following schedules. Let σ_1 be the same as σ_0 except that the messages sent by processes in P_i in the k-th layer are received by processes in $P_{(i+1) \bmod S}$ only after the k-th layer, i.e., $\sigma_1 = delay(\sigma, P_i, P_{(i+1) \bmod S}, k)$. By Lemma 10.3, we have $\alpha_0 \overset{p}{\sim} \alpha_1$, for every process $p \in P \setminus P_{(i+1) \bmod S}$. We continue inductively to define schedules as above in the following way, for every h, $0 \leq h \leq S-1$: σ_{h+1} is the same as σ_h except that the messages sent by processes in P_i in the k-th layer are received by processes in $P_{(i+h+1) \bmod S}$ only after the k-th layer, i.e., $\sigma_{h+1} = delay(\sigma_h, P_i, P_{(i+h+1) \bmod S}, k)$. By Lemma 10.3, we have $\alpha_h \overset{p}{\sim} \alpha_{h+1}$, for every process $p \in P \setminus P_{(i+h+1) \bmod S}$.

Since in σ_S no messages sent by processes in P_i in layer k are ever received. Except for local states of the processes in P_i, this is the same as if the processes in P_i are crashed in layer k:

$$\alpha_S = \alpha(crash(\sigma, P_i, k), \vec{c}, I),$$

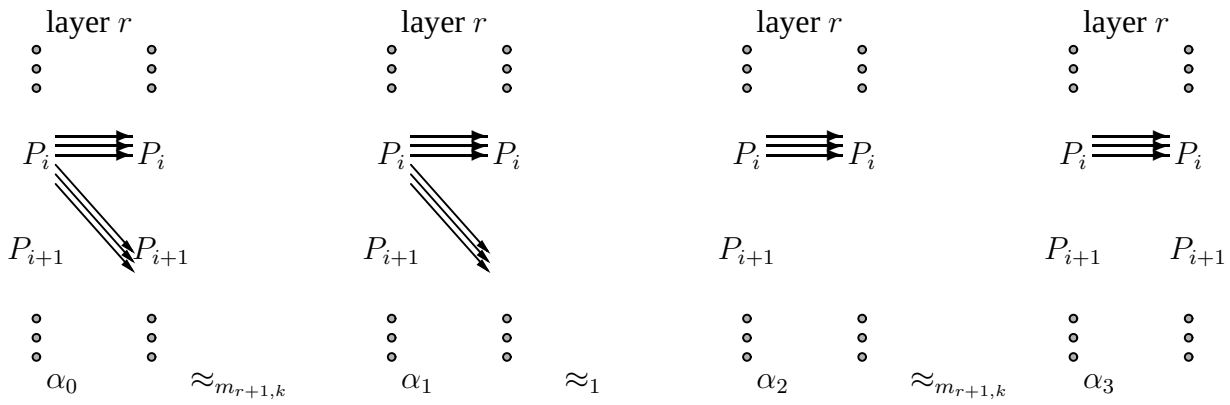


Figure 10.3: How messages from P_i to P_{i+1} are removed in the induction step of Lemma 10.4.

which implies that

$$\alpha(\sigma, \vec{c}, I) = \alpha_0 \approx_1 \alpha_1 \approx_1 \cdots \approx_1 \alpha_S = \alpha(crash(\sigma, P_i, k), \vec{c}, I) \; .$$

Therefore, $\alpha(\sigma, \vec{c}, I) \approx_S \alpha(crash(\sigma, P_i, k), \vec{c}, I)$.

Induction step: Informally, this is similar to the base case, except that we crash P_j in layer $r+1$ before "erasing" messages from P_i to P_j in layer r, and afterwards revive P_j in layer $r+1$.

Formally, we assume that the lemma holds for layer $r+1$, $1 \leq r < k$, and prove that it holds for layer r. Let $\sigma_1 = crash(\sigma_0, P_{(i+1) \bmod S}, r+1)$; by the induction hypothesis, $\alpha_0 \approx_{m_{r+1,k}} \alpha_1$.

Let σ_2 be the same as σ_1 except that the messages received by processes in $P_{(i+1) \bmod S}$ from processes in P_i in layer r are received only after the k-th layer, i.e., $\sigma_2 = delay(\sigma_1, P_i, P_{(i+1) \bmod S}, r)$. By Lemma 10.3, at the end of layer r only processes in $P_{(i+1) \bmod S}$ distinguish between the executions, but since they are crashed in layer $r+1$ we have $\alpha_1 \overset{p}{\sim} \alpha_2$, for every process $p \in P \setminus P_{(i+1) \bmod S}$, implying that $\alpha_1 \approx_1 \alpha_2$.

Let σ_3 be the same as σ_2, except that the processes in $P_{(i+1) \bmod S}$ do not crash in layer $r+1$. This implies that

$$\sigma_2 = crash(\sigma_3, P_{(i+1) \bmod S}, r+1).$$

By the induction hypothesis, we have $\alpha_2 \approx_{m_{r+1,k}} \alpha_3$. (See Figure 10.3.)

We continue inductively to define schedules as above in the following way for every h, $0 \leq h \leq S-1$. We define $\sigma_{3h+1} = crash(\sigma_{3h}, P_{(i+h+1) \bmod S}, r+1)$, and therefore by the induction hypothesis $\alpha_{3h} \approx_{m_{r+1,k}} \alpha_{3h+1}$. Let σ_{3h+2} be the same as σ_{3h+1} except that the messages received by

processes in $P_{(i+h+1) \bmod S}$ from processes in P_i in layer r are received only after the k-th layer, i.e., $\sigma_{3h+2} = delay(\sigma_{3h+1}, P_i, P_{(i+h+1) \bmod S}, r)$. By Lemma 10.3, at the end of layer r only processes in $P_{(i+h+1) \bmod S}$ distinguish between the executions, but since they are crashed in layer $r+1$ we have $\alpha_{3h+1} \overset{p}{\sim} \alpha_{3h+2}$, for every process $p \in P \setminus P_{(i+h+1) \bmod S}$, implying that $\alpha_{3h+1} \approx_1 \alpha_{3h+2}$.

Finally, we define σ_{3h+3} to be the same as σ_{3h+2}, except that processes in $P_{(i+h+1) \bmod S}$ do not crash. This implies that $\sigma_{3h+2} = crash(\sigma_{3h+3}, P_{(i+h+1) \bmod S}, r+1)$. By the induction hypothesis we have $\alpha_{3h+2} \approx_{m_{r+1,k}} \alpha_{3h+3}$.

The construction implies that in $\sigma_{3(S-1)+2}$ no messages are sent by the processes in P_i in layer r, and they are crashed from layer $r+1$. Except for local states of the processes in P_i, this is the same as if the processes in P_i are crashed from layer r. Therefore

$$\alpha(\sigma_{3(S-1)+2}, \vec{c}, I) = \alpha(crash(\sigma_0, P_i, r), \vec{c}, I),$$

and hence

$$\alpha_0 \approx_{m_{r+1,k}} \alpha_1 \approx_1 \alpha_2 \approx_{m_{r+1,k}} \alpha_3 \approx_{m_{r+1,k}} \cdots \approx_{m_{r+1,k}} \alpha_{3(S-1)+1} \approx_1 \alpha_{3(S-1)+2} .$$

Since $m_{r,k} = (2(S-1)+1)m_{r+1,k} + S$, this implies that $\alpha_0 \approx_{m_{r,k}} \alpha(crash(\sigma_0, P_i, r), \vec{c}, I)$. ■

Note that in all executions constructed in the proof, at most one set of processes P_i does not appear in a layer; since $|P_i| \leq f$, this implies that at least $n-f$ processes take a step in every layer, and hence every execution in the construction contains at least $k(n-f)$ steps.

Lemmas 10.2 and 10.4 imply that for any sequence of coins C, $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_{S(2m_{1,k}+1)} \alpha(\sigma_{full}, \vec{c}, C_S)$. Since $m_{1,k} \leq (2S)^k$, substituting $S(2m_{1,k}+1)$ in the parameter m of Theorem 10.1 yields that $q_k \geq \frac{1}{(2S)^{k+1}+S+1}$. Recall that $S = \max\{3, \lceil \frac{n}{f} \rceil\}$. Taking $\lceil \frac{n}{f} \rceil$ to be a constant, we obtain the main result of this section:

Theorem 10.5 *Let A be a randomized consensus algorithm in the asynchronous message passing model. There are a weak adversary and an initial configuration, such that the probability that A does not terminate after $k(n-f)$ steps is at least $\frac{1}{c^k}$, where c is a constant if $\lceil \frac{n}{f} \rceil$ is a constant.*

In the original construction for the synchronous model ([37, 42], see also [18, Chapter 5]), a process that does not appear in a round r must be crashed in that round, and therefore must be counted within the f failures allowed. Hence, in order to change all the inputs from 0 to 1, we

must crash and revive fewer processes at a time at each round. For example, in order to continue $k \leq f$ rounds only one process may be crashed at each round. This adds a factor of k to the base of the power in the denominator of the bound on q_k, which results in a lower bound of $\frac{1}{c \cdot k^k}$ for the synchronous message-passing model [36].

10.2 Tradeoff for the Shared-Memory Model

We now derive a similar lower bound for two shared-memory models, where processes communicate through shared read/write registers. The first model consists of single-writer registers and a cheap-snapshot operation that costs one step, described formally in Subsection 10.2.1. In Subsection 10.2.2 we consider multi-writer registers. The lower bounds clearly hold for the more restricted model, where processes read only a single register in each memory access.

10.2.1 Single-Writer Cheap-Snapshot

We first consider a shared-memory model where processes communicate through single-writer registers. The lower bound is proved under a simplifying assumption that each read step accesses the registers of all processes. We call this the *single-writer cheap-snapshot* model, since each register is written to by one specific process, and all registers are read by any process in a single snapshot.

As in a standard shared-memory model, a step of a process consists of accessing the shared memory, and performing local computations. We further assume that in the algorithm, the steps of every process alternate between a write and a cheap-snapshot, starting with a write. Any algorithm can be transformed to satisfy this requirement by having a process rewrite the same value to its register if it is forced to take a write operation, or read all of the registers and ignore some of (or all) their values if it is forced to take a cheap-snapshot operation. This only doubles the step complexity.

Recall that the processes are partitioned into $S = \max\{3, \lceil \frac{n}{f} \rceil\}$ sets $P_1, \ldots, P_S$, each with at most f processes. We consider a restricted set of layered schedules.

Definition 10.3 *A schedule σ is* regular *if for every layer L and every i, $1 \leq i \leq S$, either all processes $p \in P_i$ take a step in L consecutively (one after the other, without steps of processes not*

in P_i in between), or none of the processes $p \in P_i$ take a step in L. We denote by π the permutation of the sets P_i that take steps in L, i.e., if processes $p \in P_i$ take a step in L, then $\pi^{-1}(i)$ is their index in the layer. We denote by $|\pi|$ the number of sets P_i that take steps in the layer.

Note that, in contrast to the message-passing model, in a shared-memory model the order of the processes in a layer L is significant, since different orderings result in different executions.

Regular schedules are useful in our proofs since in every layer, all the processes in some set P_i perform the same operation, as argued in the next lemma. Since processes in the same set P_i either all write to different registers (recall that registers are single-writer) or read all registers, this means that in a regular execution, the order of processes in the set P_i does not matter.

Lemma 10.6 *Let σ be a regular schedule with k layers. Then in every layer L in σ, for every i, $1 \leq i \leq S$, either all process $p \in P_i$ do not take a step in L, or all processes $p \in P_i$ perform a write operation in L, or all processes $p \in P_i$ perform a cheap-snapshot operation in L.*

Proof: The proof is by induction on the layer number r.

Base case: Let $r = 1$, i.e., L is the first layer of σ. Since σ is regular, either all process $p \in P_i$ take a step in L, or none of the processes $p \in P_i$ take a step in L. If all take a step then by our assumption on the algorithm, it is a write operation. Otherwise, none take a step, which proves the base case.

Induction step: Assume the lemma holds for layer ℓ, $1 \leq \ell \leq r$. We prove the lemma for layer $r+1$. By the induction hypothesis, in every layer ℓ, $1 \leq \ell \leq r$, either all processes $p \in P_i$ perform a cheap-snapshot operation, or all perform a write operation, or none perform an operation. If none preform any operation in any layer $\ell \leq r$, then at the beginning of layer $r+1$ the pending operation of all processes $p \in P_i$ is a write operation by our assumption on the algorithm. Otherwise, let ℓ be the maximal layer in which all processes $p \in P_i$ took a step. If they are cheap-snapshot operations, then at the beginning of layer $r+1$ the pending operation of all processes $p \in P_i$ is a write operation by our assumption on the algorithm. If they are write operations, then at the beginning of layer $r+1$ the pending operation of all processes $p \in P_i$ is a cheap-snapshot operation by our assumption on the algorithm. In any case, at the beginning of layer $r+1$, either all processes $p \in P_i$ have a pending cheap-snapshot operation, or all have a pending write operation. Since σ is regular, either none of the processes $p \in P_i$ take a step in layer $r+1$, or all take a step in layer $r+1$, in which

case it would either be a cheap-snapshot operation for all processes, or a write operation for all processes. ■

In the proof, we apply certain manipulations to regular schedules, allowing us to delay and crash sets of processes, as follows.

Definition 10.4 *Let σ be a schedule such that every $p \in P_i$ is non-faulty in layer r, and such that P_i is not the last set of processes in the layer. Let $swap(\sigma, P_i, r)$ be the schedule that is the same as σ, except that the steps of processes in P_i are swapped with steps of the next set of processes in that layer. Formally, if π is the permutation of layer r in σ and π' is the permutation of layer r in $swap(\sigma, P_i, r)$, and if $j = \pi^{-1}(i)$, then we have $\pi'(j) = \pi(j+1)$ and $\pi'(j+1) = \pi(j)$.*

Inductively, we define

$$swap^j(\sigma, P_i, r) = swap(swap^{j-1}(\sigma, P_i, r), P_i, r),$$

that is, P_i is swapped j times and moved j sets later in the layer.

Definition 10.5 *Let σ be a schedule and r be a layer such that no process is skipped in any layer $\ell > r$. Let $delay(\sigma, P_i, r)$ be the schedule that is the same as σ, except that the steps of P_i starting from layer r are delayed by one layer. Thus, there is no step of $p \in P_i$ in layer r, the step of $p \in P_i$ in layer $r+1$ is the step that was in layer r, and so on. The permutations of the layers $\ell \geq r+1$ do not change.*

Note that this definition assumes a schedule σ in which no process is skipped in any layer $\ell > r$. Specifically, this implies that P_i appears in every layer $\ell \geq r+1$, which allows to keep the permutations in layers $\ell \geq r+1$ unchanged in $delay(\sigma, P_i, r)$.

Delaying a set P_i from layer r can be seen as delaying P_i from layer $r+1$, swapping P_i in layer r until it reaches the end of the layer, accounting for P_i as the first set in layer $r+1$ instead of the last set in layer r, and then swapping P_i in layer $r+1$ until it reaches its original place in the layer.

Although accounting for P_i as the first set in layer $r+1$ instead of the last set in layer r does not change the order of steps taken, it is technically a different schedule (recall that the schedules are defined as sequences of layers, which in this case are different in layers r and $r+1$). Therefore we define:

Definition 10.6 *Let σ be a schedule where the last set of processes in layer r is P_i, and this set does not appear in layer $r+1$. Let $rollover(\sigma, P_i, r)$ be the schedule that is the same as σ, except that P_i is the first set in layer $r+1$ instead of the last set in layer r.*

Effectively, such two schedules σ and $rollover(\sigma, P_i, r)$ have the same order of steps, which implies that the executions of these schedules is the same:

$$\alpha(\sigma, \vec{c}, I) = \alpha(rollover(\sigma, P_i, r), \vec{c}, I).$$

Definitions 10.4, 10.5, and 10.6 imply:

Corollary 10.7 *Let σ be a regular schedule with k layers, and for every r, $1 \leq r \leq k$, let π_r be the permutation of layer r in σ. Then,*

$$delay(\sigma, P_i, r) = swap^{\pi_{r+1}^{-1}(i)-1}(rollover(swap^{|\pi_r|-\pi_r^{-1}(i)}(delay(\sigma, P_i, r{+}1), P_i, r), P_i, r), P_i, r{+}1).$$

Figure 10.4 depicts the schedules used when delaying a set P_i in layer r of a schedule σ, according to this corollary.

Recall that $crash(\sigma, P_i, r)$ is the schedule that is the same as σ, except that processes in P_i crash in layer r. Crashing a set P_i in layer r can be seen as delaying it from layer r, and then crashing it from layer $r+1$. Definitions 10.1 and 10.5 imply that:

Corollary 10.8 *For every regular schedule σ,*

$$crash(\sigma, P_i, r) = crash(delay(\sigma, P_i, r), P_i, r+1).$$

An important property of regular schedules is that swapping, delaying, or crashing a set of processes P_i yields a regular schedule as well, because the sets are manipulated together.

Lemma 10.9 *Let σ be a regular schedule with k layers. Then for every i, $1 \leq i \leq S$, and every r, $1 \leq r \leq k$, the schedules $swap(\sigma, P_i, r)$, $delay(\sigma, P_i, r)$, $rollover(\sigma, P_i, r)$, and $crash(\sigma, P_i, r)$ are regular.*

Proof: Every layer $\ell \neq r$ in $swap(\sigma, P_i, r)$ is the same as in σ and therefore satisfies the requirement of a regular schedule. In layer r, all processes that took steps in σ also take steps in

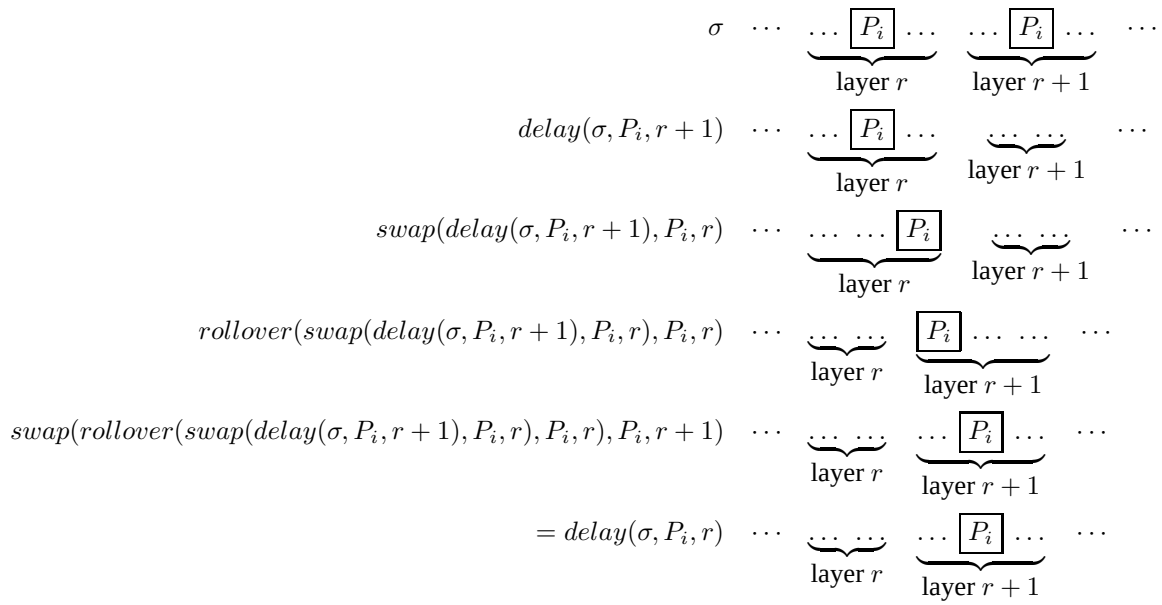


Figure 10.4: An example showing how $swap$ operators are applied to delay a set of processes P_i; assume P_i is the penultimate set in layer r and the third set in layer $r+1$. Note that the third transition does not modify the execution, and only accounts the steps of P_i to layer $r+1$ instead of layer r; the last transition just notes that we have obtained $delay(\sigma, P_i, r)$.

$swap(\sigma, P_i, r)$, and each set remains consecutive. Therefore, $swap(\sigma, P_i, r)$ is regular. It is also easy to see that $rollover(\sigma, P_i, r)$ is regular.

The proof for $delay(\sigma, P_i, r)$ and $crash(\sigma, P_i, r)$ is by backwards induction on the layer number r.

Base case: For $r = k$, delaying a set P_i in the last layer, is the same as crashing P_i. Denote $\sigma' = delay(\sigma, P_i, k) = crash(\sigma, P_i, k)$. Every layer $\ell < k$ in σ' is the same as in σ, and the last layer k is the same in σ' except that the processes in P_i do not take a step. Hence, σ' is also regular.

Induction step: We assume the lemma holds for every layer ℓ, $r+1 \leq \ell \leq k$, and prove it for layer r. By Corollary 10.7, the induction hypothesis and since swapping results in a regular schedule, $delay(\sigma, P_i, r)$ is regular. By Corollary 10.8, the induction hypothesis and since delaying results in a regular schedule, $crash(\sigma, P_i, r)$ is regular. ■

We next construct an indistinguishability chain of schedules between any regular schedule and a schedule in which some set of processes is delayed or crashed. The construction relies on

Corollary 10.7 and Corollary 10.8 to delay or crash a set of processes through a sequence of swaps. The elementary step in this construction, where a set is swapped with the following one, is provided by the next lemma.

Lemma 10.10 *Let σ be a regular schedule with k layers. For any sequences of coins $\vec{c}$, and initial configuration I, if P_i is not the last set of processes in layer r, $1 \leq r \leq k$, then there is a set P_j such that at the end of layer r only processes in P_j (at most) distinguish between $\alpha(\sigma, \vec{c}, I)$ and $\alpha(swap(\sigma, P_i, r), \vec{c}, I)$.*

Proof: Consider $swap(\sigma, P_i, r)$ and let π be the permutation corresponding to layer r. Since P_i is not the last set in the layer, we have $\pi^{-1}(i) \neq |\pi|$. Let $i' = \pi(\pi^{-1}(i) + 1)$, i.e., P_i is swapped with $P_{i'}$. By Lemma 10.6, either all the processes in P_i perform a cheap-snapshot operation or all processes in P_i perform a write operation. The same applies for $P_{i'}$.

Let C be the configuration resulting from both executions after layer $r-1$ and $\vec{y}_1$ the vector of the resulting coin flips of the processes at the configuration C. Further, let L be layer r in σ, and L' be layer r in $swap(\sigma, P_i, r)$. By Claim 9.1, there is a set P_j such that the configurations at the end of layer r of both executions are indistinguishable to processes not in P_j (where j is either i or i'). ■

Notice that the set P_j (the value of the index j) depends only on the types of operations performed, i.e, only on σ, and not on the sequences of coins $\vec{c}$ or the initial configuration I. This is crucial for ensuring that the adversary is non-adaptive.

For every r and k, $1 \leq r \leq k$, we define:

$$s_{r,k} = \begin{cases} 1 & \text{if } r = k \\ 2 \cdot c_{r+1,k} + 1 & \text{if } 1 \leq r < k \end{cases}$$

$$d_{r,k} = \begin{cases} S & \text{if } r = k \\ d_{r+1,k} + S \cdot s_{r,k} + S \cdot s_{r+1,k} & \text{if } 1 \leq r < k \end{cases}$$

$$c_{r,k} = \begin{cases} S & \text{if } r = k \\ d_{r,k} + c_{r+1,k} & \text{if } 1 \leq r < k \end{cases}$$

where $S = \max\{3, \lceil \frac{n}{f} \rceil\}$ is the number of sets P_i. These recursive functions will be used for bounding the lengths of the indistinguishability chains in our construction.

The next proposition shows a bound on these functions; its proof is a simple backwards induction.

Proposition 10.11 $c_{r,k} \leq (2S+4)^{k-r+1}$.

The main technical result of this section is the following lemma, which will be used to show an indistinguishability chain between the executions that result from schedules σ and $crash(\sigma, P_i, 1)$, in order to apply Lemma 10.2. Additional claims, regarding $swap$ and $delay$, are proved in order to carry through with the proof.

Lemma 10.12 *Let σ be a regular schedule with k layers such that no process is skipped at any layer $\ell \geq r$, for some r, $1 \leq r \leq k$. For any sequences of coins $\vec{c}$, and initial configuration I, and for every i, $1 \leq i \leq S$, the following all hold:*

$$\begin{aligned}
\alpha(\sigma, \vec{c}, I) &\approx_{s_{r,k}} \alpha(swap(\sigma, P_i, r), \vec{c}, I), \\
\alpha(\sigma, \vec{c}, I) &\approx_{d_{r,k}} \alpha(delay(\sigma, P_i, r), \vec{c}, I), \\
\alpha(\sigma, \vec{c}, I) &\approx_{c_{r,k}} \alpha(crash(\sigma, P_i, r), \vec{c}, I).
\end{aligned}$$

Proof: Let $\sigma_0 = \sigma$. Throughout the proof, we denote $\alpha_i = \alpha(\sigma_i, \vec{c}, I)$ for every schedule σ_i, and $\alpha'_i = \alpha(\sigma'_i, \vec{c}, I)$ for every schedule σ'_i.

The proof is by backwards induction on r.

Base case: $r = k$. Consider $swap(\sigma, P_i, k)$, where P_i is not the last set in the layer (otherwise swapping is undefined). By Lemma 10.10, there is a set P_j, which does not depend on $\vec{c}$ or I, such that $\alpha(\sigma, \vec{c}, I) \overset{p}{\sim} \alpha(swap(\sigma, P_i, r), \vec{c}, I)$, for every process $p \notin P_j$. Therefore, $\alpha(\sigma, \vec{c}, I) \approx_{s_{k,k}} \alpha(swap(\sigma, P_i, k), \vec{c}, I)$.

Delaying P_i in the last layer is equivalent to failing it, therefore $delay(\sigma, P_i, k) = crash(\sigma, P_i, k)$. Denote this schedule by σ'. We crash P_i by swapping it until it reaches the end of the layer and then removing it. In more detail, let π be the permutation of the last layer of σ, and define:

$$\sigma'' = swap^{|\pi| - \pi^{-1}(i)}(\sigma, P_i, k).$$

The proof of the base case for $swap(\sigma, P_i, k)$ implies that there is a chain of length $s_{k,k} \cdot (|\pi_r| - \pi_r^{-1}(i)) \leq (S-1) \cdot s_{k,k} = S - 1$ between the executions, i.e., $\alpha_0 \approx_{S-1} \alpha(\sigma'', \vec{c}, I)$.

Clearly, $\alpha(\sigma'', \vec{c}, I) \overset{p}{\sim} \alpha(\sigma', \vec{c}, I)$, for every process $p \notin P_i$, and therefore, $\alpha(\sigma, \vec{c}, I) \approx_{d_{k,k}} \alpha(delay(\sigma, P_i, r), \vec{c}, I)$ and $\alpha(\sigma, \vec{c}, I) \approx_{c_{k,k}} \alpha(crash(\sigma, P_i, r), \vec{c}, I)$.

Induction step: Assume the lemma holds for layer $r + 1 \leq k$. We prove that it holds for layer r.

We first deal with swapping; assume that P_i is not the last set in the layer and consider $swap(\sigma, P_i, r)$. By Lemma 10.10, there is a set P_j, which does not depend on $\vec{c}$ or I, such that at the end of layer r only process in P_j distinguish between $\alpha(\sigma, \vec{c}, I)$ and $\alpha(swap(\sigma, P_i, r), \vec{c}, I)$. We define σ_1 to be the same as σ except that processes in P_j are crashed in layer $r + 1$, i.e., $\sigma_1 = crash(\sigma, P_j, r + 1)$. By the induction hypothesis, $\alpha_0 \approx_{c_{r+1,k}} \alpha_1$. Let σ_2 be the same as σ_1 except that P_i and P_j are swapped in layer r, i.e., $\sigma_2 = swap(\sigma_1, P_i, r)$. Since only processes in P_j observe the swapping, but are all crashed in the next layer, we have that $\alpha_1 \overset{p}{\sim} \alpha_2$ for every process $p \notin P_j$. Finally, let σ_3 be the same as σ_2, except that processes in P_j are not crashed in layer $r + 1$, i.e., $\sigma_2 = crash(\sigma_3, P_j, r + 1)$. By the induction hypothesis, $\alpha_2 \approx_{c_{r+1,k}} \alpha_3$. Notice that $\sigma_3 = swap(\sigma, P_i, r)$, and $2c_{r+1,k} + 1 = s_{r,k}$, which implies that $\alpha(\sigma, \vec{c}, I) \approx_{s_{r,k}} \alpha(swap(\sigma, P_i, r), \vec{c}, I)$.

Next, we consider the case of delaying a process, i.e., $delay(\sigma, P_i, r))$. (Recall Figure 10.4.) By Corollary 10.7,

$$delay(\sigma, P_i, r) = swap^{\pi_{r+1}^{-1}(i)-1}(rollover(swap^{|\pi_r|-\pi_r^{-1}(i)}(delay(\sigma, P_i, r{+}1), P_i, r), P_i, r), P_i, r{+}1).$$

Recall that applying $rollover$ does not change the execution. Hence, by the proof for swapping, the induction hypothesis, and since

$$d_{r+1,k} + s_{r,k} \cdot (|\pi_r| - \pi_r^{-1}(i)) + s_{r+1,k} \cdot (\pi_{r+1}^{-1}(i) - 1) \leq d_{r+1,k} + S \cdot s_{r,k} + S \cdot s_{r+1,k} = d_{r,k}$$

it follows that $\alpha(\sigma, \vec{c}, I) \approx_{d_{r,k}} \alpha(delay(\sigma, P_i, r), \vec{c}, I)$.

Finally, we consider the case of crashing a process, i.e., $crash(\sigma, P_i, r)$. By Corollary 10.8,

$$crash(\sigma, P_i, r) = crash(delay(\sigma, P_i, r), P_i, r + 1).$$

By the proof for delaying, the induction hypothesis, and since $d_{r,k} + c_{r+1,k} = c_{r,k}$, it follows that

$$\alpha(\sigma, \vec{c}, I) \approx_{c_{r,k}} \alpha(crash(\sigma, P_i, r), \vec{c}, I).$$

■

Note that in all executions constructed in the proof, at most one set of processes P_i does not appear in a layer; since $|P_i| \leq f$, this implies that at least $n - f$ processes take a step in every layer, and hence every execution in the construction contains at least $k(n - f)$ steps.

Lemmas 10.2 and 10.12 imply that for every sequence of coins $\vec{c}$, $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_{S(2c_{1,k}+1)+1} \alpha(\sigma_{full}, \vec{c}, C_S)$, Since $c_{1,k} \leq (2S+4)^k$, we have that $S(2c_{1,k}+1)+1 \leq (2S+4)^{k+1}$. Substituting in Theorem 10.1 yields that $q_k \geq \frac{1}{(2S+4)^{k+1}+1}$. Since S can be taken to be a constant when $\lceil \frac{n}{f} \rceil$ is a constant, we get the next theorem:

Theorem 10.13 *Let A be a randomized consensus algorithm in the asynchronous shared-memory model, with single-writer registers and cheap snapshots. There are a weak adversary and an initial configuration, such that the probability that A does not terminate after $k(n - f)$ steps is at least $\frac{1}{c^k}$, where c is a constant if $\lceil \frac{n}{f} \rceil$ is a constant.*

10.2.2 Multi-Writer Cheap-Snapshot

We derive the lower bound for multi-writer registers by reduction to single-writer registers. In a simple simulation of a multi-writer register from single-writer registers (e.g., [73]), performing a high-level read or write operation (to the multi-writer register) involves n low-level read operations (of all single-writer registers) and possibly one low-level write operation (to the process' own single-writer register). This multiplies the total step complexity by $O(n)$.

However, with cheap-snapshots, we can read all single-writer registers in one step, yielding a simulation that only doubles the total step complexity (since writing includes a cheap-snapshot operation). The pseudocode of the simulation appears in Algorithm 10.1, which uses an array RR of single-writer variables. $RR[i]$ is the last value written by p_i, together with a timestamp. The correctness of this algorithm follows along the proof of Algorithm 10.3 from [18].

Since in the single-writer cheap-snapshot model each snapshot operation accounts for one access to the shared memory, every algorithm in the multi-writer model can be transformed to an algorithm in the single-writer cheap-snapshot model, where the step complexity is only multiplied by a constant factor. Combining this with Theorem 10.13 yields the next theorem.

Theorem 10.14 *Let A be a randomized consensus algorithm in the asynchronous shared-memory model, with multi-writer registers and cheap snapshots. There are a weak adversary and an initial*

Algorithm 10.1 Simulating a multi-writer register R from single-writer registers.

Process p_i has a shared register $RR[1]$, each consisting of the pair $\langle v, t \rangle$;
initially, each register holds $\langle 0, init \rangle$, where $init$ is the desired initial value

1: $read(R)$:
2: snapshot RR array into local (t, v) array
3: return $v[j]$ such that $t[j]$ is maximal

4: $write(R, v)$ by p_w:
5: snapshot RR array into local (t, v) array
6: let lts be the maximum of $t[1], \ldots, t[n]$
7: write the pair $(v, lts + 1)$ to $RR[w]$
8: return

configuration, such that the probability that A does not terminate after $k(n-f)$ steps is at least $\frac{1}{c^k}$, where c is a constant if $\lceil \frac{n}{f} \rceil$ is a constant.

10.3 Monte-Carlo Algorithms

Another way to overcome the impossibility of asynchronous consensus is to allow Monte-Carlo algorithms. This requires us to relax the agreement property and allow the algorithm to decide on conflicting values, with small probability. Let ϵ_k be the maximum probability, over all weak adversaries and over all initial configurations, that processes decide on conflicting values after $k(n-f)$ steps. The next theorem is the analogue of Theorem 10.1, for bounding the probability of terminating after $k(n-f)$ steps.

Theorem 10.15 *Assume there is an integer m such that for all sequences of coins $\vec{c}$, $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$. Then $q_k \geq \frac{1-(m+1)\epsilon_k}{m+1}$.*

Proof: Assume, by way of contradiction, that $(m+1)q_k < 1 - (m+1)\epsilon_k$. Consider the $m+1$ executions in the sequence implied by the fact that $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$. The probability that A does not terminate in at least one of these $m+1$ executions is at most $(m+1)q_k$.

By assumption, $q_k(m+1) < 1-(m+1)\epsilon_k$, and hence, the set B of sequences of coins $\vec{c}$ such that A terminates in all $m+1$ executions has probability $\Pr[\vec{c} \in B] > (m+1)\epsilon_k$.

If the agreement property is satisfied in all $m+1$ executions, then by the validity condition, as in the proof of Theorem 10.1, we get that the decision in $\alpha(\sigma_{full}, \vec{c}, C_0)$ is the same as the decision in $\alpha(\sigma_{full}, \vec{c}, C_S)$, which is a contradiction. Hence, for every $\vec{c} \in B$, the agreement condition does not hold in at least one of these executions.

Since we have $m+1$ schedules in the chain, there exists a schedule for which the agreement condition does not hold with probability greater than ϵ_k. But this means that A satisfies agreement with probability smaller than $1-\epsilon_k$, which is a contradiction. ■

Substituting with Lemma 10.2 and Lemma 10.4, yields the lower bound for the message passing model.

Theorem 10.16 *Let A be a randomized consensus algorithm in the asynchronous message passing model. There are a weak adversary and an initial configuration, such that the probability that A does not terminate after $k(n-f)$ steps is at least $\frac{1-c^k\epsilon_k}{c^k}$, where c is a constant if $\lceil \frac{n}{f} \rceil$ is a constant, and ϵ_k is a bound on the probability for disagreement.*

Substituting with Lemma 10.2 and Theorem 10.14 yields the lower bound for the shared memory model.

Theorem 10.17 *Let A be a randomized consensus algorithm in the asynchronous shared-memory model with multi-writer registers and cheap snapshots. There are a weak adversary and an initial configuration, such that the probability that A does not terminate after $k(n-f)$ steps is at least $\frac{1-c^k\epsilon_k}{c^k}$, where c is a constant if $\lceil \frac{n}{f} \rceil$ is a constant, and ϵ_k is a bound on the probability for disagreement.*

The bound we obtain in Theorem 10.15 on the probability q_k of not terminating increases as the allowed probability ϵ_k of terminating without agreement decreases, and coincides with Theorem 10.1 in case the agreement property must always be satisfied (i.e., $\epsilon_k = 0$). In case an algorithm always terminates in $k(n-f)$ steps (i.e., $q_k = 0$), we can restate Theorem 10.15 as a bound on ϵ_k:

Corollary 10.18 *Assume there is an integer m such that for all sequences of coins $\vec{c}$, $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$. Moreover, assume that the algorithm always terminates after $k(n-f)$ steps, i.e., $q_k = 0$. Then $\epsilon_k \geq \frac{1}{m+1}$.*

For example, the algorithms for the message-passing model given by Kapron et al. [52] are Monte-Carlo, i.e., have a small probability for terminating without agreement. They present an algorithm that always terminates within $\mathrm{polylog}(n)$ asynchronous rounds, and has a probability $\frac{1}{\mathrm{polylog}(n)}$ for disagreeing. For comparison, our lower bound of Corollary 10.18 for disagreeing when $k = \mathrm{polylog}(n)$ and $q_k = 0$ is $\epsilon_k \geq \frac{1}{c^{\mathrm{polylog}(n)}}$, where c is a constant if $\lceil \frac{n}{f} \rceil$ is a constant. Their second algorithm always terminates within $2^{\Theta(\log^7 n)}$ asynchronous rounds, and has a probability $\frac{1}{\mathrm{poly}(n)}$ for disagreeing, while our lower bound is $\epsilon_k \geq \frac{1}{c^{2^{\Theta(\log^7 n)}}}$.

Chapter 11

A Lower Bound for a Strong Adversary

In this section we prove a lower bound for randomized consensus under a strong adversary, as defined in Section 9.2. We begin, in Section 11.1, by laying the setting for the framework we use. This section defines the key notions used in our lower bound proof—*potence* and *valence*—and proves that layered executions in the multi-writer shared-memory model are *potence connected*. The lower bound proof appears in Section 11.2.

We emphasize that we are considering multi-writer registers, which we could not have assumed when manipulating layers for the lower bound under the weak adversary because there the register written to by a process has to be fixed in advance for the adversary to be non-adaptive. However, under a strong adversary we cannot directly employ the reduction of Section 10.2.2 from single-writer cheap-snapshot registers to multi-writer registers because the adaptiveness of our strong adversary may imply a weakening of the algorithm (splitting operations that are modelled as atomic operations into several memory accesses). Dealing with multi-writer registers directly imposes some subtle challenges when proving indistinguishability of configurations, therefore the manipulations we perform on the layers for proving the lower bound in this section are more involved. Moreover, coping with these manipulations requires additional definitions which are specific for this lower bound, and hence do not appear in Chapter 9.

11.1 Potence and Valence of Configurations

In order to derive our lower bound, we are interested in the probability of reaching each of the possible decision values, from a given configuration. As the algorithm proceeds towards a decision, we expect the probability of reaching a decision to grow, where for a configuration in which a decision is reached, this probability is 1. This intuition is formalized as follows. Let $k \geq 0$ be an integer, and define

$$\epsilon_k = \frac{1}{n\sqrt{n}} - \frac{k}{(n-f)^3}.$$

Our proof makes use of adversaries that have a probability of $1 - \epsilon_k$ for reaching a certain decision value from a configuration reached after k layers. As the layer number k increases, the value of ϵ_k decreases, and the probability $1 - \epsilon_k$ required for a decision grows. The value of ϵ_k is set with foresight to achieve the stated bound.

An adversary with a high probability of deciding is defined as follows.

Definition 11.1 *An f-adversary α from a configuration C that is reachable from an initial configuration by an f-execution with $k \geq 0$ layers, is* v-deciding *if Pr[decision from C under α is v] $> 1 - \epsilon_k$.*

Next, we classify configurations according to the probabilities of reaching each of the possible decisions from them. We adapt the notion of *potence* [60] to fit randomized algorithms.

Instead of considering all possible adversaries, we further restrict our attention to a certain subset of them, which will be specified later.

Definition 11.2 *A configuration C that is reachable from an initial configuration by an f-execution with $k \geq 0$ layers, is* (v, k, S)-potent, *for $v \in \{0, 1\}$ and a set S of f-adversaries, if there is a v-deciding adversary $\alpha \in S$ from C.*

Definition 11.3 *A configuration is* (v, k, S)-valent *if it is (v, k, S)-potent but not $(\bar{v}, k, S)$-potent. Such a configuration is also called* (k, S)-univalent.

A configuration is (k, S)-bivalent *if it is both $(0, k, S)$-potent and $(1, k, S)$-potent.*

A configuration is (k, S)-null-valent *if it is neither $(0, k, S)$-potent nor $(1, k, S)$-potent.*

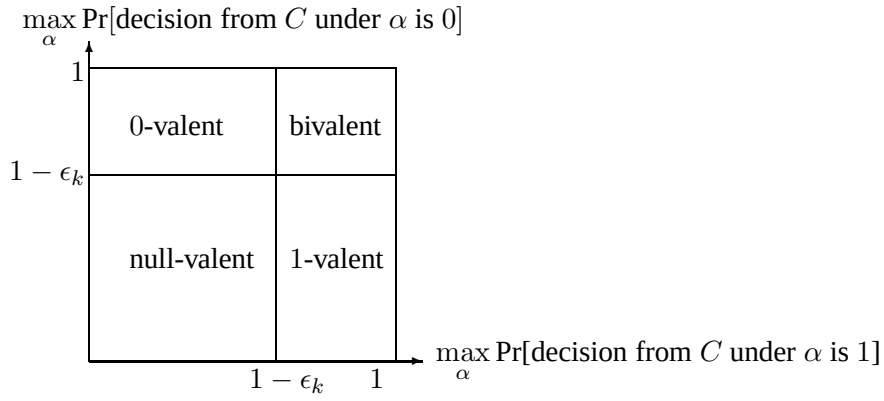


Figure 11.1: Classifying configurations according to their valence.

We often say that C is v-potent (v-valent, bivalent, null-valent) *with respect to* S, when the number of layers k is clear from the context. Further, if the set S is also clear from the context, we will sometimes use the notation v-potent (v-valent, bivalent, null-valent). Figure 11.1 illustrates the valence of configurations as follows. A configuration C is mapped to a point in the figure, according to the maximum probabilities over all adversaries for deciding 0 and for deciding 1 from C; the valence of C is determined by the area in which the respective point lies. For example, if this point lies beyond $1 - \epsilon_k$ on the x axis, it implies that there is an adversary σ from C with probability at least $1 - \epsilon_k$ for deciding 1. Therefore, C is either bivalent or 1-valent, depending on whether it lies beyond $1 - \epsilon_k$ on the y axis or not.

Note that a configuration can have a certain valence with respect to one set of adversaries S and another valence with respect to a different set S'. For example, it can be univalent with respect to S and null-valent with respect to $S' \neq S$; however, this cannot happen when $S \subseteq S'$. (Another example appears in Lemma 11.1 below.)

The set of f-adversaries we consider, denoted S_P, is induced by a subset of processes $P \subseteq \{p_1, \ldots, p_n\}$. An adversary α is in S_P, if all of the layers it may choose are P-free, where a layer is *P-free* if it does not include any process $p \in P$. Specifically, the set $S_\emptyset$ is the set of all f-adversaries.

A layer is *full with respect to* S_P if it contains the $n - |P|$ distinct process identifiers $\{p_1, \ldots, p_n\} \setminus P$; otherwise, the layer is *partial*. Scheduling a partial layer without some process $p \notin P$, does not imply that p is failed, since the system is asynchronous, and p may appear in

later layers. However, considering only adversaries in S_P for some nonempty set P, is equivalent to failing the processes in P since they do not take further steps.

Restricting a set of adversaries can only eliminate possible adversaries, and therefore cannot introduce potence that does not exist in the original set of adversaries, as formalized in the following simple lemma.

Lemma 11.1 *If a configuration C, reached after k layers, is v-valent with respect to S_P, then it is not $\bar{v}$-potent with respect to $S_{P\cup\{p\}}$ for any process p.*

Proof: Assume towards a contradiction, that there is a process p such that C is $\bar{v}$-potent with respect to $S_{P\cup\{p\}}$. Then there exists a $\bar{v}$-deciding adversary α in $S_{P\cup\{p\}}$, i.e.,

$$\Pr[\text{decision in } C' \circ \alpha \text{ is } \bar{v}] > 1 - \epsilon_k.$$

But α is also an adversary in S_P because $S_{P\cup\{p\}} \subseteq S_P$, which implies that C is $\bar{v}$-potent also with respect to S_P, contradicting the fact that C is v-valent with respect to S_P. ■

Let C be a configuration reached after some number of layers k, and fix a vector $\vec{y}_1 \in X^C$. We consider every configuration that can be reached by applying a single layer to C. We define a relation between these various configurations, based on their potence, which generalizes notions for deterministic algorithms suggested by [60].

Definition 11.4 *For a given vector $\vec{y}_1$, two configurations $(C, \vec{y}_1, L)$ and $(C, \vec{y}_1, L')$ have* shared potence with respect to S_P*, if they are both v-potent with respect to S_P for some $v \in \{0, 1\}$.*

We define *potence connectivity* between two layers, as the transitive closure of the above relation.

Definition 11.5 *For a given vector $\vec{y}_1$, two configurations $(C, \vec{y}_1, L)$ and $(C, \vec{y}_1, L')$ are* potence connected with respect to S_P*, if there is a sequence of layers $L = L_0, L_1, \ldots, L_h = L'$ such that for every i, $0 \leq i < h$, there exists a process p such that the configurations $(C, \vec{y}_1, L_i)$ and $(C, \vec{y}_1, L_{i+1})$ have shared potence with respect to $S_{P\cup\{p\}}$.*

In particular, if $(C, \vec{y}_1, L)$ and $(C, \vec{y}_1, L')$ have shared potence with respect to $S_{P\cup\{p\}}$ for some process p, then they are potence connected.

Our goal is to show that given C, $\vec{y_1}$ and S_P, if the set of all configurations of the form $(C, \vec{y}_1, L)$ does not contain a null-valent configuration, then these configurations are potence connected with respect to S_P. Therefore, if there are both 0-potent and 1-potent configurations in this set, then there must also be a bivalent configuration in it. This would imply that there is a non-univalent configuration among this set, namely, a configuration that is not v-valent, for any v.

The following claims are used to prove this connectivity, by showing that specific configurations are potence connected. They are proved under the following assumption:

Assumption 1 *Let C be a configuration, $\vec{y_1} \in X^C$, and S_P a set of adversaries. For every process p and every layer L, the configuration $(C, \vec{y_1}, L)$ is univalent with respect to S_P and $S_{P\cup\{p\}}$.*

We proceed to stating and proving our connectivity claims.

Claim 11.2 *Under Assumption 1, if $L = [p_{i_1}, p_{i_2}, \ldots, p_{i_\ell}]$ is a layer where for some j, $1 \leq j < \ell$, p_{i_j} and $p_{i_{j+1}}$ both write to the same register R, and $L' = [p_{i_1}, \ldots, p_{i_{j-1}}, p_{i_{j+1}}, \ldots, p_{i_\ell}]$ is the layer L after removing p_{i_j}, then $(C, \vec{y}_1, L)$ and $(C, \vec{y}_1, L')$ have shared potence with respect to $S_{P\cup\{p_{i_j}\}}$.*

Proof: By Claim 9.2, taking each set to be a single process, we have that $(C, \vec{y}_1, L) \overset{\neg P\cup\{p_{i_j}\}}{\sim} (C, \vec{y}_1, L')$, which implies that $(C, \vec{y}_1, L)$ and $(C, \vec{y}_1, L')$ have the same potence with respect to $S_{P\cup\{p_{i_j}\}}$. By Assumption 1, this implies that they have shared potence with respect to $S_{P\cup\{p_{i_j}\}}$, since they are not null-valent. ■

Claim 11.3 *Under Assumption 1, if $L = [p_{i_1}, p_{i_2}, \ldots, p_{i_\ell}]$ is a layer, p is a process not in L, and $L' = [p_{i_1}, p_{i_2}, \ldots, p_{i_\ell}, p]$ is the layer L after adding p at the end, then $(C, \vec{y}_1, L)$ and $(C, \vec{y}_1, L')$ have shared potence with respect to $S_{P\cup\{p\}}$.*

Proof: If p performs a read operation, then $(C, \vec{y}_1, L) \overset{\neg P\cup\{p\}}{\sim} (C, \vec{y}_1, L')$, which implies that these two configurations have shared potence with respect to $S_{P\cup\{p\}}$, and the claim follows.

If p performs a write operation to register R, then the states of all processes not in $P \cup \{p\}$ are the same in $(C, \vec{y}_1, L)$ and in $(C, \vec{y}_1, L')$, but the value of R may be different.

If $(C, \vec{y}_1, L)$ and $(C, \vec{y}_1, L')$ do not have shared potence with respect to $S_{P\cup\{p\}}$, then since we assume they are univalent with respect to $S_{P\cup\{p\}}$ (Assumption 1), we have that for some $v \in \{0, 1\}$,

$(C, \vec{y}_1, L)$ is v-valent with respect to $S_{P\cup\{p\}}$ and $(C, \vec{y}_1, L')$ is $\bar{v}$-valent with respect to $S_{P\cup\{p\}}$. In particular, there is a $\bar{v}$-deciding adversary $\alpha \in S_{P\cup\{p\}}$ from $(C, \vec{y}_1, L')$. Adding p at the beginning of the first layer of α yields a new adversary which is in S_P, and is $\bar{v}$-deciding when applied to $(C, \vec{y}_1, L)$, since it is the same as applying α to $(C, \vec{y}_1, L')$. However, by Lemma 11.1, $(C, \vec{y}_1, L)$ is v-valent with respect to S_P, which is a contradiction. ■

Claim 11.4 *Under Assumption 1, if* $L = [p_{i_1}, p_{i_2}, \ldots, p_{i_\ell}]$ *is a layer and* $L' = [p_{i_1}, \ldots, p_{i_{j-1}}, p_{i_{j+1}}, p_{i_j}, p_{i_{j+2}}, \ldots, p_{i_\ell}]$ *is the layer* L *after swapping* p_{i_j} *and* $p_{i_{j+1}}$*, then* $(C, \vec{y}_1, L)$ *and* $(C, \vec{y}_1, L')$ *are potence connected with respect to* S_P*.*

Proof: By Claim 9.1, taking each set to be a single process, if p_{i_j} and $p_{i_{j+1}}$ access different registers, or if they both read, or if p_{i_j} reads register R and $p_{i_{j+1}}$ writes to R or vice versa, then

$$(C, \vec{y}_1, L) \overset{\neg P\cup\{p\}}{\sim} (C, \vec{y}_1, L') ,$$

where p is either p_{i_j} or $p_{i_{j+1}}$. Both cases imply that $(C, \vec{y}_1, L)$ and $(C, \vec{y}_1, L')$ are potence connected with respect to S_P. The remaining case is when p_{i_j} and $p_{i_{j+1}}$ both write to the same register R. It may be that all the rest of the processes in the layer read from register R, and therefore distinguish between the two resulting configurations. Hence, we cannot argue in this case that the configurations have shared potence, but we can prove that they are potence connected. This is done by reverse induction on j.

Basis: If $j = \ell - 1$, let $L_0 = L$, $L_1 = [p_{i_1}, \ldots, p_{i_{\ell-2}}, p_{i_\ell}]$ be the layer L after removing $p_{i_{\ell-1}}$, and $L_2 = L'$. By Claim 11.2, $(C, \vec{y}_1, L_0)$ and $(C, \vec{y}_1, L_1)$ are potence connected with respect to S_P, and by Claim 11.3, $(C, \vec{y}_1, L_1)$ and $(C, \vec{y}_1, L_2)$ are potence connected with respect to S_P. The transitivity of potence connectivity implies that $(C, \vec{y}_1, L_0)$ and $(C, \vec{y}_1, L_2)$ are potence connected with respect to S_P.

Induction step: Let

$$L_0 = L = [p_{i_1}, p_{i_2}, \ldots, p_{i_\ell}]$$

and let

$$L_1 = [p_{i_1}, \ldots, p_{i_{j-1}}, p_{i_{j+1}}, p_{i_{j+2}}, \ldots, p_{i_\ell}]$$

be the layer L_0 after removing p_{i_j}. By Claim 11.2, $(C, \vec{y}_1, L_0)$ and $(C, \vec{y}_1, L_1)$ are potence connected with respect to S_P. Let

$$L_2 = [p_{i_1}, \ldots, p_{i_{j-1}}, p_{i_{j+1}}, p_{i_{j+2}}, \ldots, p_{i_\ell}, p_{i_j}]$$

L_0 =	p_1:r(R_1)	...	p_i:w(R_2)	p_{i+1}:r(R_2)	p_{i+2}:w(R_2)	...	p_{n-1}:r(R_3)	p_n:r(R_4)
L' =	p_1:r(R_1)	...	p_i:w(R_2)	p_{i+1}:r(R_2)	p_{i+2}:w(R_2)	...	p_{n-1}:r(R_3)	
L'' =	p_1:r(R_1)	...	p_{i+1}:r(R_2)	p_i:w(R_2)	p_{i+2}:w(R_2)	...	p_{n-1}:r(R_3)	
L_1 =	p_1:r(R_1)	...	p_{i+1}:r(R_2)		p_{i+2}:w(R_2)	...	p_{n-1}:r(R_3)	

Figure 11.2: Example of potence connected configurations: the first and second configuration are connected by Claim 11.3, the second and third are connected by Claim 11.4, while the third and fourth are connected by Claim 11.2.

be the layer L_1 after adding p_{i_j} at the end. By Claim 11.3, $(C, \vec{y}_1, L_1)$ and $(C, \vec{y}_1, L_2)$ are potence connected with respect to S_P.

For every m, $3 \leq m \leq \ell - j + 1$, let

$$L_m = [p_{i_1}, \ldots, p_{i_{j-1}}, p_{i_{j+1}}, p_{i_{j+2}}, \ldots, p_{i_j}, p_{i_{\ell-m+3}}, \ldots, p_{i_\ell}]$$

be the previous layer L_{m-1} after swapping p_{i_j} with the process before it, until it reaches $p_{i_{j+1}}$. Specifically,

$$L_{\ell-j+1} = L' = [p_{i_1}, \ldots, p_{i_{j-1}}, p_{i_{j+1}}, p_{i_j}, p_{i_{j+2}}, \ldots, p_{i_\ell}].$$

By the induction hypothesis, $(C, \vec{y}_1, L_m)$ and $(C, \vec{y}_1, L_{m+1})$ are potence connected with respect to S_P, for every m, $2 \leq m < \ell - j + 1$. The transitivity of potence connectivity implies that $(C, \vec{y}_1, L_0)$ and $(C, \vec{y}_1, L_{\ell-j+1})$ are potence connected with respect to S_P. ■

Figure 11.2 shows an example of using the claims, for the full layer L_0 and the partial layer L_1 obtained by removing the steps of p_i and p_n from L_0. We show two layers L' and L'' such that only one process, p_n, distinguishes between the configurations lead to by L_0 and L', only one process, p_{i+1}, distinguishes between the configurations lead to by L' and L'', and only one process, p_i, distinguishes between the configurations lead to by L'' and L_1. Thus, the configurations lead to by L_0 and L_1 must be potence connected.

The following lemma shows that given a configuration C and a vector $\vec{y} \in X^C$, if there is a layer that extends C into a v-valent configuration and a layer that extends C into a $\bar{v}$-valent configuration, then we can find a layer that extends C into a non-univalent configuration, possibly by failing one additional process.

Lemma 11.5 *Let C be a configuration and let $\vec{y}_1$ be a vector in X^C. If there are layers L_v and $L_{\bar{v}}$, such that $(C, \vec{y}_1, L_v)$ is $(v, k+1, S_P)$-valent and $(C, \vec{y}_1, L_{\bar{v}})$ is $(\bar{v}, k+1, S_P)$-valent, then there is a layer L such that $(C, \vec{y}_1, L)$ is either not $(k+1, S_P)$-univalent or not $(k+1, S_{P\cup\{p\}})$-univalent, for some process p.*

Proof: Assume towards a contradiction that for every layer L and every process p, the configuration $(C, \vec{y}_1, L)$ is univalent with respect to both S_P and $S_{P\cup\{p\}}$ (this implies that Assumption 1 holds). Let L^F be the full layer with respect to S_P consisting of all processes not in P, according to the order of their id's. Then, L^F is univalent with respect to S_P, say it is $(\bar{v}, k+1, S_P)$-valent. (Otherwise, we follow the same proof with $L_{\bar{v}}$.)

Denote $L_v = [p_{i_1}, \ldots, p_{i_\ell}]$ and consider the layer $L' = [p_{i_1}, \ldots, p_{i_\ell}, \ldots]$ that is full with respect to S_P, and has L_v as a prefix. (L_v may be full with respect to S_P, in which case L' is equal to L_v.)

We start with the layer L^F and repeatedly swap processes until we reach the layer L', in a chain of configurations which, by Claim 11.4, are potence connected with respect to S_P. From L', we repeatedly remove the last process until reaching the layer L_v, in a chain of configurations which, by Claim 11.3, are potence connected with respect to S_P. This implies that $(C, \vec{y}_1, L^F)$ and $(C, \vec{y}_1, L_v)$ are potence connected with respect to S_P.

Since $(C, \vec{y}_1, L_v)$ is $(v, k+1, S_P)$-valent, and $(C, \vec{y}_1, L^F)$ is $(\bar{v}, k+1, S_P)$-valent, it follows that there are layers L_1 and L_2 such that $(C, \vec{y}_1, L_1)$ is $(v, k+1, S_P)$-valent, $(C, \vec{y}_1, L_2)$ is $(\bar{v}, k+1, S_P)$-valent, and $(C, \vec{y}_1, L_1)$ and $(C, \vec{y}_1, L_2)$ have shared potence with respect to $S_{P\cup\{p\}}$, for some process p. By Lemma 11.1 and our assumption that all layers lead to univalent configurations, $(C, \vec{y}_1, L_1)$ is $(v, k+1, S_{P\cup\{p\}})$-valent, and $(C, \vec{y}_1, L_2)$ is $(\bar{v}, k+1, S_{P\cup\{p\}})$-valent, and hence, they cannot have shared potence with respect to $S_{P\cup\{p\}}$. This yields a contradiction and proves the lemma. ■

11.2 The Lower Bound

Before presenting the lower bound proof, let us recall lower bound proofs and impossibility results for deterministic consensus algorithms. In these proofs, the configurations are classified into *univalent* and *bivalent* configurations. Since a deciding configuration has to be univalent, these proofs aim to avoid univalent configurations by showing that there is an initial bivalent configuration, and by moving from one bivalent configuration to another bivalent configuration.

Our proof generalizes the above technique to randomized algorithms as follows. Recall that in addition to bivalent and univalent configurations, we have null-valent configurations, since valence is now a probabilistic notion. We first show that some initial configuration is not univalent (Lemma 11.6); namely, it is either bivalent or null-valent.

Ideally, we would like to complete the proof by showing that a non-univalent configuration can be extended by a single layer to a non-univalent configuration, while (permanently) failing at most one more process. Doing so would allow us to construct a layered execution with f layers, each containing at least $n - f$ process steps, which implies the desired lower bound.

In Lemma 11.11 (Section 11.2.4), we show that this can be done with high probability in the case of null-valent configurations, i.e., we can extend a null-valent configuration by one layer and reach another null-valent configuration.

A bivalent configuration, can be extended with both a v-deciding adversary and a $\bar{v}$-deciding adversary, which we would like to use in Lemma 11.5 to obtain a non-univalent configuration. However, some complications arise here, which are taken care of in Lemmas 11.7 and 11.8 (Section 11.2.2).

We extend the execution in this manner, with high probability, for f layers. Since $n - f$ processes take a step in each layer, we obtain the bound of an expected $\Omega(f(n - f))$ steps (Theorem 11.12).

11.2.1 Initial Configurations

We start by applying Lemma 11.1 to show that some initial configuration is not univalent.

Lemma 11.6 *There exists an initial configuration C that is not univalent with respect to either $S_\emptyset$ or $S_{\{p\}}$, for some process p.*

Proof: Assume that all initial configurations are univalent with respect to $S_\emptyset$. Consider the initial configurations $C_0, C_1, \ldots, C_n$ such that in C_i, $0 \leq i \leq n$, the input of process p_j is 1 if $j \leq i$ and 0, otherwise. By the validity condition, C_0 is $(0, 0, \emptyset)$-valent and C_n is $(1, 0, \emptyset)$-valent. Therefore, there is an i, $1 \leq i \leq n$, such that C_{i-1} is $(0, 0, \emptyset)$-valent and C_i is $(1, 0, \emptyset)$-valent. Clearly, $C_{i-1} \overset{\neg p_i}{\sim} C_i$, and hence, C_{i-1} and C_i have the same valence with respect to $S_{\{p_i\}}$. By Lemma 11.1, C_{i-1} is not 1-potent with respect to $S_{\{p_i\}}$, and C_i is not 0-potent with respect to $S_{\{p_i\}}$. Hence, they are null-valent with respect to $S_{\{p_i\}}$. ■

11.2.2 Bivalent and Switching Configurations

As mentioned before, from a bivalent configuration we have both a v-deciding adversary and a $\bar{v}$-deciding adversary. However, we cannot use them directly in Lemma 11.5 to obtain a non-univalent configuration, since the first layer of a v-deciding adversary may still lead to a $\bar{v}$-valent configuration, because these definitions are probabilistic. If ϵ_k was an increasing function of k, the above situation would have small enough probability in order to simply neglect it. However, this cannot be done, since ϵ_k is defined as a decreasing function of k, in order to handle the null-valent configurations.

Instead, we prove (Lemma 11.7) that by failing at most one more process, a bivalent configuration can be extended by a single layer to a non-univalent configuration, or to a configuration which is $\bar{v}$-valent although reached while following a v-deciding adversary. Such a configuration, as will be formalized below, is called *$\bar{v}$-switching*.

We also prove that there is a small probability of deciding in a switching configuration and thus, from a switching configuration, the execution can be extended (with high probability) to a non-univalent configuration, by at least one layer (Lemma 11.8).

We formally define switching configurations as follows.

Definition 11.6 *Let C be a (v, k, S_P)-potent configuration, let $\alpha = \sigma_1, \sigma_2, \ldots$ be a v-deciding adversary from C in S_P, and let $\vec{y}_1$ be a vector in X^C such that the configuration $(C, \vec{y}_1, \sigma_1(\vec{y}_1))$ is $(\bar{v}, k+1, S_P)$-valent. Then $(C, \vec{y}_1, \sigma_1(\vec{y}_1))$ is a* $\bar{v}$-switching configuration with respect to S_P from C by $\vec{y}_1$ and α.

Lemma 11.7 implies that a bivalent configuration can be extended with one layer to a configuration that is either switching or non-univalent.

Lemma 11.7 *If a configuration C is (k, S_P)-bivalent, then there is an adversary σ such that for every vector $\vec{y}_1 \in X^C$, $(C, \vec{y}_1, \sigma(\vec{y}_1))$ is either $\bar{v}$-switching for some $v \in \{0, 1\}$, or not $(k+1, S_P)$-univalent or not $(k+1, S_{P \cup \{p\}})$-univalent, for some process p.*

Proof: Assume that for every layer L and every process p, the configuration $(C, \vec{y}_1, L)$ is univalent with respect to S_P and to $S_{P \cup \{p\}}$.

Consider the extension of C with L^F, the full layer with respect to S_P. Fix a vector $\vec{y}_1 \in X^C$ and assume that $C'' = (C, \vec{y}_1, L^F)$ is $(\bar{v}, k+1, S_P)$-valent. Since C is bivalent, there is a v-deciding adversary $\alpha = \sigma_1, \sigma_2, \ldots$ in S_P. Consider the configuration $C' = (C, \vec{y}_1, \sigma_1(\vec{y}_1))$. By the assumption, C' is univalent. If it is $(\bar{v}, k+1, S_P)$-valent then it is $\bar{v}$-switching with respect to S_P from C by $\vec{y}_1$ and α. Otherwise, it is $(v, k+1, S_P)$-valent. Since C'' is $(\bar{v}, k+1, S_P)$-valent, by Lemma 11.5, there exists a layer L and a process p such that $(C, \vec{y}_1, L)$ is either not $(k+1, S_P)$-univalent or not $(k+1, S_{P \cup \{p\}})$-univalent. ■

The main issue when reaching a $\bar{v}$-switching configuration is that we cannot rule out the possibility that all the other layers lead to $\bar{v}$-valent configurations as well. However, although a $\bar{v}$-switching configuration is $\bar{v}$-valent, the adversary that leads to it is v-deciding. This allows us to look several layers ahead, for the desired situation of one layer leading to a v-valent configuration, and another layer leading to a $\bar{v}$-valent configuration. From such a setting, Lemma 11.5 can be used to reach a non-univalent configuration again. The following lemma presents the details of this argument.

Lemma 11.8 *Let C' be a $\bar{v}$-switching configuration with respect to S_P from C by $\vec{y}_1$ and α. Then with probability at least $1 - \frac{1}{n\sqrt{n}}$, C' can be extended with at least one layer to a configuration which is either v-switching, or not univalent with respect to S_P, or not univalent with respect to $S_{P \cup \{p\}}$, for some process p.*

Proof: Let $\alpha = \sigma_1 \sigma_2 \ldots$; note that $C' = (C, \vec{y}_1, \sigma_1(\vec{y}_1))$. Denote $C_0 = C$, $C_1 = C'$ and for every $k \geq 2$, fix a vector $\vec{y}_k \in X^{C_{k-1}}$ and let $C_k = (C_{k-1}, \vec{y}_k, \sigma_k(\vec{y}_k))$.

Assume that for every $k \geq 1$, C_k is univalent, and let C_ℓ be the first configuration which is v-valent with respect to S_P. If such a configuration does not exist then the execution either reaches a configuration that decides $\bar{v}$, or does not reach any decision. Since α is v-deciding from C, the probability that C_ℓ does not exist is at most $\epsilon_k \leq \frac{1}{n\sqrt{n}}$.

Since C_ℓ is the first configuration which is v-valent, $C_{\ell-1}$ is $\bar{v}$-valent. Therefore, there is a $\bar{v}$-deciding adversary $\beta = \rho_1, \rho_2, \ldots$ from $C_{\ell-1}$ in S_P. Consider the configuration $C'' = (C_{\ell-1}, \vec{y}_\ell, \rho_1(\vec{y}_\ell))$. If C'' is not $(k+\ell, S_P)$-univalent then we are done. If C'' is $\bar{v}$-valent, then since C_ℓ is v-valent, by Lemma 11.5, there exists a layer L such that $(C_{\ell-1}, \vec{y}_\ell, L)$ is either not $(k+\ell, S_P)$-univalent or not $(k+\ell, S_{P \cup \{p\}})$-univalent, for some process p, in which case we are

also done. Otherwise C'' is v-valent, which implies that it is v-switching with respect to S_P from $C_{\ell-1}$ by $\vec{y}_\ell$ and β. ■

11.2.3 One-Round Coin-Flipping Games

The remaining part of the lower bound proof deals with null-valent configurations, and it relies on results about one-round coin-flipping games. As defined in [23], a *U-valued one-round coin flipping game of m players* is a function

$$g : \{X_1 \cup \perp\} \times \{X_2 \cup \perp\} \times \cdots \times \{X_m \cup \perp\} \longrightarrow \{1, 2, \ldots, U\},$$

where X_i, $1 \leq i \leq m$, is the i-th probability space. A *t-hiding adversary* may hide at most t of the random choices in $X_1, \ldots, X_m$, by replacing them with a $\perp$.

Before presenting the formal definitions for the adversary and the game, we describe one main difference in the way that null-valent and bivalent configurations are handled. We have shown in Section 11.2.2 that in order to reach a bivalent configuration from a bivalent configuration, it suffices to fail one process per layer. This process is failed permanently, and may not take steps in any later layer. The case of a null-valent configuration is different. We may need to hide many more processes in a layer in order to reach another null-valent configuration. Fortunately, these processes are only *hidden* in this layer and may take steps in subsequent layers, which implies that they are not failed according to the definition of an asynchronous system. Therefore, we do not need to account for them towards the f processes that we are allowed to fail, and hence their number can be non-constant.

Formally, let $X = X_1 \times \cdots \times X_m$ be the product probability space implied by the game. For every vector $\vec{y} \in X$, and $I \subseteq \{1, \ldots, m\}$, the vector reached when the adversary hides the coordinates of I is defined as follows:

$$\vec{y}_I(i) = \begin{cases} \vec{y}(i), & i \notin I \\ \\ \perp, & i \in I. \end{cases}$$

For every possible outcome of the game $u \in \{1, \ldots, U\}$, define the set of all vectors in X for which no t-hiding adversary can force the outcome of the game to be u, to be

$$W^u = \{\vec{y} \in X | g(\vec{y}_I) \neq u \text{ for every } I \subseteq \{1, \ldots, m\} \text{ such that } |I| \leq t \}.$$

We prove that when $|U| = 3$, there is an outcome for which there is high probability for hiding values in a way that forces this outcome; that is, for some $u \in \{1, 2, 3\}$, Pr$[\vec{y} \in W^u]$ is very small. The proof relies on an isoperimetric inequality, following a result of [69]. The idea is to show that if all three sets $W^1, W^2, W^3 \subseteq X$ have large enough probability, then there is a non-zero probability for an element $\vec{z} \in X$ that is close to all three sets according to the Hamming distance. This will imply the existence of a t-hiding adversary for $\vec{z}$, for which the value of the game cannot be any of 1, 2 or 3, and hence there is a point for which the game is undefined.

Formally, the space (X, d) is a finite metric space where for every pair of vectors $\vec{x}$ and $\vec{y}$ in X, $d(\vec{x}, \vec{y})$ is the Hamming distance between $\vec{x}$ and $\vec{y}$ (the number of coordinates in which $\vec{x}$ and $\vec{y}$ differ). For $A \subseteq X$, $B(A, t)$ is the *ball of radius t around the set A*, i.e.,

$$B(A, t) = \{\vec{y} \in X | \text{ there is } \vec{z} \in A \text{ such that } d(\vec{y}, \vec{z}) \leq t\}.$$

The next lemma is the isoperimetric inequality we rely on. We show that given the probability of a set A, there is a lower bound on the probability of the ball of a certain radius around A.

Lemma 11.9 *Let $X = X_1 \times \cdots \times X_m$ be a product probability space and $A \subseteq X$ such that Pr$[\vec{x} \in A] = c$. Let $\lambda_0 = \sqrt{2m \log \frac{2}{c}}$, then for $\ell \geq \lambda_0$,*

$$Pr[\vec{x} \in B(A, \ell)] \geq 1 - 2e^{-\frac{(\ell - \lambda_0)^2}{2m}}.$$

Proof: Consider an element $\vec{x}$ as a random function $\vec{x} : D = \{1, \ldots, m\} \rightarrow X_1 \cup \cdots \cup X_m$ such that $\vec{x}(i) \in X_i$. Define a sequence of partial domains $\emptyset = D_0 \subset D_1 \subset \cdots \subset D_m = D$ such that $D_i = \{1, \ldots, i\}$. Let $f : X \rightarrow \mathbb{R}$ be a function that measures the distance of elements from the given subset $A \subseteq X$, i.e., $f(\vec{x}) = d(A, \vec{x})$.

Choose a random element of $\vec{w} \in X$ according to the given distribution. Define the sequence $Y_0, \ldots, Y_m$ by $Y_i = \mathrm{E}[f(\vec{x}) \mid \vec{x}|_{D_i} = \vec{w}]$. Specifically, $Y_0 = \mathrm{E}[f(\vec{x})]$ with probability 1, and $Y_m = f(\vec{w})$ with the probability of choosing $\vec{w}$. It is well known that $Y_0, \ldots, Y_m$ is a Doob martingale (see [3, Chapter 7] and [63, Chapter 4]).

Notice that $X_1, ..., X_m$ are independent and therefore for every $i \in D$, the random variable $\vec{x}(i)$ is independent of other values of $\vec{x}$.

The function f satisfies the Lipschitz condition, i.e., for every 2 vectors $\vec{x}, \vec{y}$ that differ only in $D_i \setminus D_{i-1}$ for some i, we have $|f(\vec{x}) - f(\vec{y})| \leq 1$, by the triangle inequality of the Hamming

metric d. This implies that the martingale $Y_0, \ldots, Y_m$ satisfies the martingale Lipschitz condition, i.e., $|Y_i - Y_{i-1}| \leq 1$ for every i, $0 < i \leq m$.

By Azuma's inequality we have that for every real number $\lambda > 0$

$$\Pr[|f(\vec{x}) - \mathrm{E}[f(\vec{x})]| > \lambda] < 2e^{-\frac{\lambda^2}{2m}}. \tag{11.1}$$

We now claim that $\mathrm{E}[f(\vec{x})] \leq \lambda_0$. Assume the contrary, that $\mathrm{E}[f(\vec{x})] > \lambda_0$. Since $\lambda_0 = \sqrt{2m \log \frac{2}{c}}$, we have that $2e^{-\frac{\lambda_0^2}{2m}} = c$. For every $\vec{x} \in A$ we have $f(\vec{x}) = 0$, therefore

$$\Pr[|f(\vec{x}) - \mathrm{E}[f(\vec{x})]| > \lambda_0] \geq \Pr[f(\vec{x}) = 0] = c,$$

contradicting (11.1).

Hence, for every $\ell \geq \lambda_0$ we have

$$\Pr[\vec{x} \notin B(A, \ell)] = \Pr[f(\vec{x}) > \ell] \leq \Pr[|f(\vec{x}) - \mathrm{E}[f(\vec{x})]| > \ell - \lambda_0] < 2e^{-\frac{(\ell - \lambda_0)^2}{2m}},$$

which completes the proof. ∎

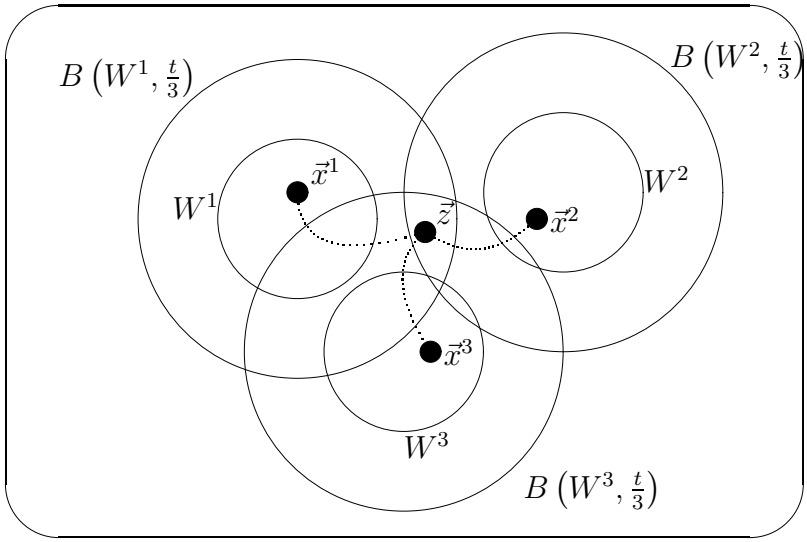
We now use Lemma 11.9 to show that one of the sets W^u has a small probability, and deduce that for $u = 1, 2$ or 3 the outcome of the game can be forced to be u with high probability.

Lemma 11.10 *For every 3-valued one-round coin-flipping game of m players, there is a t-hiding adversary, $t = 6\sqrt{2m \log(m^3)}$, that can force the outcome of the game to be some $u \in \{1, 2, 3\}$ with probability greater than $1 - \frac{1}{m^3}$.*

Proof: Recall that for every $u \in \{1, 2, 3\}$, the set W^u is the set of all vectors in X for which no t-hiding adversary can force the outcome of the game to be u. We wish to prove that $\Pr[\vec{y} \in W^u] < \frac{1}{m^3}$, for some $u \in \{1, 2, 3\}$.

Denote $B^u = B(W^u, \frac{t}{3})$. Assume that $\Pr[\vec{y} \in W^u] \geq \frac{1}{m^3}$ for all $u \in \{1, 2, 3\}$. Clearly, $\bigcap_{u \in \{1,2,3\}} W^u = \emptyset$, since the value of the game is undefined for points in the intersection. Moreover, we claim that $\bigcap_{u \in \{1,2,3\}} B^u$ is empty.

Assume there is a point $\vec{z} \in \bigcap_{u \in \{1,2,3\}} B^u$ (see Figure 11.3). For every $u \in \{1, 2, 3\}$, since $\vec{z} \in B^u$, there is a point $\vec{x}^u \in W^u$ and a set of indices $I_u \subseteq \{1, \ldots, m\}$, such that $|I_u| \leq \frac{t}{3}$ and $\vec{z}_{I_u} = \vec{x}^u_{I_u}$. Let $I = \bigcup_{u \in \{1,2,3\}} I_u$. Since $\vec{x}^u \in W^u$, we have that $g(\vec{x}^u_I) \neq u$, and hence $g(\vec{z}_I) \neq u$. This implies that $g(\vec{z}_I)$ is undefined, and therefore $\bigcap_{u \in \{1,2,3\}} B^u = \emptyset$.



$$g(\vec{z}_s) = g(\vec{x}^1_s) = g(\vec{x}^2_s) = g(\vec{x}^3_s)$$

$$\vec{z}_s = \boxed{\perp}\boxed{\perp}\boxed{\perp}\boxed{\ }\boxed{\ }\boxed{\perp}\boxed{\ }\boxed{\ }\boxed{\perp}\boxed{\ }$$

$$\vec{x}^1_{s_1} = \boxed{\ }\boxed{\perp}\boxed{\perp}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$$

$$\vec{x}^2_{s_2} = \boxed{\perp}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\perp}\boxed{\ }$$

$$\vec{x}^3_{s_3} = \boxed{\ }\boxed{\ }\boxed{\perp}\boxed{\ }\boxed{\ }\boxed{\perp}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$$

Figure 11.3: The probability space $X = X_1 \times X_2 \times \cdots \times X_m$. The distance between the point $\vec{z}$ to each of the points $\vec{x}^1, \vec{x}^2, \vec{x}^3$ is at most $\frac{t}{3}$.

We now apply Lemma 11.9 for every $u = 1, 2, 3$, with $A = W^u$. Notice that the results of the local flips of each player are independent random variables. We have that $\Pr[\vec{y} \in W^u] \geq c$ where $c = \frac{1}{m^3}$, and therefore $\lambda_0 = \sqrt{2m \log(2m^3)}$. Hence, for $\ell = \frac{t}{3} = 2\sqrt{2m \log(2m^3)} = 2\lambda_0$, we have

$$\Pr[\vec{y} \in B^u] \geq 1 - 2e^{-\frac{(t-\lambda_0)^2}{2m}} = 1 - 2e^{-\frac{2m \log(2m^3)}{2m}} = 1 - 2e^{-\log(2m^3)} = 1 - \frac{1}{m^3}.$$

Since $\bigcap_{u \in \{1,2,3\}} B^u = \emptyset$ we have that

$$\Pr[\vec{y} \in B^1 \cap B^2] + \Pr[\vec{y} \in B^1 \cap B^3] + \Pr[\vec{y} \in B^2 \cap B^3] \leq \frac{1}{2} \cdot \sum_{u \in \{1,2,3\}} \Pr[\vec{y} \in B^u],$$

which implies that

$$\begin{aligned} \Pr[\vec{y} \in \cup_{u \in \{1,2,3\}} B^u] &= \sum_{u \in \{1,2,3\}} \Pr[\vec{y} \in B^u] - \sum_{u \neq u' \in \{1,2,3\}} \Pr[\vec{y} \in B^u \cap B^{u'}] + \Pr[\vec{y} \in \bigcap_{u \in \{1,2,3\}} B^u] \\ &\geq \frac{1}{2} \cdot \sum_{u \in \{1,2,3\}} \Pr[\vec{y} \in B^u] \\ &\geq \frac{3}{2} \cdot (1 - \frac{1}{m^3}) > 1. \end{aligned}$$

This contradiction implies that for some $u \in \{1, 2, 3\}$, we have $\Pr[\vec{y} \in W^u] < \frac{1}{m^3}$. ■

11.2.4 Null-Valent Configurations

In the final stage of the lower bound construction, we use one-round coin-flipping games to show that, with high probability, a null-valent configuration C can be extended with one f-layer to a null-valent configuration. In order to achieve the above, we may need to hide up to $6\sqrt{2n \log(2n^3)}$ processes, other than the processes in P, in the layer. Therefore, we assume that $f \geq 12\sqrt{2n \log(2n^3)}$, and will always make sure that $|P| \leq \frac{f}{2}$. This will allow us to hide $\frac{f}{2} \geq 6\sqrt{2n \log(2n^3)}$ additional processes (not in P), in executions in S_P.

Lemma 11.11 *If a configuration C reachable by an f-execution is (k, S_P)-null-valent, then with probability at least $1 - \frac{1}{(n-|P|)^3}$, there is an f-adversary σ_1 such that $C \circ \sigma_1$ is $(k+1, S_P)$-null-valent.*

Proof: Let C be a (k, S_P)-null-valent configuration. We consider every vector $\vec{y}_1 \in X^C$ and every layer L in S_P, and classify the resulting configurations $(C, \vec{y}_1, L)$ into three disjoint categories:

1. The configuration $(C, \vec{y}_1, L)$ is $(0, k+1, S_P)$-potent.

2. The configuration $(C, \vec{y}_1, L)$ is $(1, k+1, S_P)$-valent.

3. The configuration $(C, \vec{y}_1, L)$ is $(k+1, S_P)$-null-valent.

Notice that the first category contains both $(0, k+1, S_P)$-valent and $(k+1, S_P)$-bivalent configurations.

This can be viewed as a 3-valued one-round coin-flipping game of m players, as follows. The m players are the processes not in S_P, i.e., $m = n - |P|$. The probability spaces $X_1, \ldots, X_m$ represent the random choices of the processes, which are given by $\vec{y_1}$. Every vector of choices induces a resulting configuration $(C, \vec{y_1}, L^F)$, where L^F is the full layer with respect to S_P, in which the processes take a step in the order of their identifiers. Hiding an element X_i by the adversary is done by choosing a partial layer L in S_P that does not contain any step by the corresponding processes, but only a step of each process that is not hidden. Finally, the value of the game is the category of the configuration $(C, \vec{y}_1, L)$.

Since $m = n - |P|$, we have that $n - f \leq m \leq n$. By the coin flipping game in Lemma 11.10, we can hide $6\sqrt{2m \log(2m^3)}$ processes and force the resulting configuration into one of the above categories with probability at least $1 - \frac{1}{m^3}$.

This implies that for one of the above categories, with probability at least $1 - \frac{1}{m^3}$, the vector $\vec{y}_1 \in X^C$ has a corresponding partial layer $L_{\vec{y}_1}$, such that the configuration $(C, \vec{y}_1, L_{\vec{y}_1})$ has the valence of that category. We now define the adversary σ_1 as the function that for every vector $\vec{y}_1 \in X^C$ chooses the corresponding partial layer $L_{\vec{y}_1}$, i.e., $\sigma_1(\vec{y_1}) = L_{\vec{y}_1}$. Our claim is that the category that can be forced by σ_1 is the third one, i.e., the resulting configuration is null-valent.

Assume, towards a contradiction, that the category that can be forced is the first one. This implies that the probability over all vectors $\vec{y}_1 \in X^C$ that $(C, \vec{y}_1, L_{\vec{y}_1})$ is $(0, k+1, S_P)$-potent is at least $1 - \frac{1}{m^3}$. Therefore, with probability at least $1 - \frac{1}{m^3}$, the vector $\vec{y}_1 \in X^C$ is such that there exists a 0-deciding adversary α' from $(C, \vec{y}_1, L_{\vec{y}_1})$ for which:

$$\Pr[\text{decision from } (C, \vec{y}_1, L_{\vec{y}_1}) \text{ under } \alpha' \text{ is } 0] > 1 - \epsilon_{k+1}$$

Therefore with probability at least $1 - \frac{1}{m^3}$, there exists an adversary $\alpha = \sigma_1, \alpha'$ from C such that:

Pr[decision from C under α is 0] =

$$
\begin{aligned}
&= \sum_{\vec{y}_1 \in X^C} \Pr[\vec{y}_1] \cdot \Pr[\text{decision from } (C, \vec{y}_1, L_{\vec{y}_1}) \text{ under } \alpha' \text{ is } 0] \\
&> \left(1 - \frac{1}{m^3}\right) \cdot (1 - \epsilon_{k+1}) \\
&\geq \left(1 - \frac{1}{(n-f)^3}\right) \cdot \left(1 - \frac{1}{n\sqrt{n}} + \frac{k+1}{(n-f)^3}\right) \\
&= 1 - \frac{1}{n\sqrt{n}} + \frac{k}{(n-f)^3} + \frac{1}{(n-f)^3 n\sqrt{n}} - \frac{k+1}{(n-f)^6} \\
&> 1 - \frac{1}{n\sqrt{n}} + \frac{k}{(n-f)^3} = 1 - \epsilon_k,
\end{aligned}
$$

where the last inequality holds for sufficiently large n, since $(n-f)^6 \geq (n-f)^3 n\sqrt{n}$ and $k = O(n)$. This contradicts the assumption that C is (k, S_P)-null-valent.

A similar argument holds for the second category. Hence, with probability at least $1 - \frac{1}{m^3}$, the third category can be forced, namely, we can reach a configuration that is $(k+1, S_P)$-null-valent.

■

11.2.5 Putting the Pieces Together

We can now put the pieces together and prove the lower bound on the total step complexity of any randomized consensus algorithm.

Theorem 11.12 *The total step complexity of any f-tolerant randomized consensus algorithm in an asynchronous system, where $n - f \in \Omega(n)$ and $f \geq 12\sqrt{2n \log(2n^3)}$, is $\Omega(f(n-f))$.*

Proof: We show that the probability of forcing the algorithm to continue $\frac{f}{2}$ layers is at least $1 - \frac{1}{\sqrt{n}}$. Therefore the expected number of layers is at least $(1 - \frac{1}{\sqrt{n}}) \cdot \frac{f}{2}$. Each of these layers is an f-layer containing at least $n - f$ steps, implying that the expected total number of steps is at least $\Omega((1 - \frac{1}{\sqrt{n}}) \cdot \frac{f}{2} \cdot (n-f))$, which is in $\Omega(f(n-f))$ since $n - f \in \Omega(n)$.

We argue that for every k, $0 \leq k \leq \frac{f}{2}$, with probability at least $1 - k\frac{1}{n\sqrt{n}}$, there is a configuration C reachable by an f-execution with at least k layers, which is either v-switching or non-univalent with respect to S_P where $|P| \leq k+1$. Once the claim is proved, the theorem follows by taking

$k = \frac{f}{2}$, since the probability of having an f-execution with more than $\frac{f}{2}$ layers is at least $1 - \frac{f}{2} \cdot \frac{1}{n\sqrt{n}} > 1 - \frac{1}{\sqrt{n}}$.

We prove the claim by induction on k.

Basis: $k = 0$. By Lemma 11.6, there exists an initial configuration C that is not univalent with respect to $S_\emptyset$ or $S_{\{p\}}$, for some process p.

Induction step: Assume C is a configuration reachable by an f-execution with at least k layers, that is either v-switching or non-univalent with respect to S_P where $|P| \leq k + 1$. We prove that with probability at least $1 - \frac{1}{n\sqrt{n}}$, C can be extended with at least one layer to a configuration C' that is either v-switching or non-univalent with respect to $S_{P'}$ where $|P'| \leq k + 2$. This implies that C' exists with probability $(1 - k\frac{1}{n\sqrt{n}})(1 - \frac{1}{n\sqrt{n}}) \geq 1 - (k+1)\frac{1}{n\sqrt{n}}$.

If C is bivalent, then by Lemma 11.7, there exists an adversary σ and a process p such that $C \circ \sigma$ is either v-switching or not $(k+1, S_P)$-univalent or not $(k+1, S_{P\cup\{p\}})$-univalent.

If C is v-switching, then by Lemma 11.8, there exists a finite adversary α_ℓ and a process p such that with probability at least $1 - \frac{1}{n\sqrt{n}}$, $C \circ \alpha_\ell$ is either $\bar{v}$-switching, or not univalent with respect to S_P, or not univalent with respect to $S_{P\cup\{p\}}$.

If C is null-valent, then by Lemma 11.11, there exists an adversary σ_1 such that the configuration $C \circ \sigma_1$ is not $(k+1, S_P)$-univalent with probability at least $1 - \frac{1}{m^3}$. Since $m \geq n - f \in \Omega(n)$, we have that $1 - \frac{1}{m^3} \geq 1 - \frac{1}{(n-f)^3} > 1 - \frac{1}{n\sqrt{n}}$. ■

Finally, taking $f \in \Omega(n)$ and $n - f \in \Omega(n)$, we get a lower bound of $\Omega(n^2)$ on the total step complexity.

Chapter 12

Discussion

This thesis studies several problems in distributed computing, motivated by the classic problem of randomized consensus. Probabilistic methods are used to construct and analyze randomized algorithms for consensus, shared coins, counters and additional concurrent data structures, and set agreement, as well as for deriving lower bounds for randomized consensus.

We have shown, in Chapter 4 and in Chapter 11, that $\Theta(n^2)$ is a tight bound on the total step complexity of solving randomized consensus, under a strong adversary, in an asynchronous shared-memory system using multi-writer registers.

Our algorithm exploits the multi-writing capabilities of the register. The best known randomized consensus algorithm using single-writer registers [28] has $O(n^2 \log n)$ total step complexity, and it is intriguing to close the gap from our lower bound. Note that for the single-writer cheap-snapshot model, $\Theta(n^2)$ is a tight bound, since our lower bound applies to it, and a slight modification of our algorithm yields the same total step complexity. This is done by replacing the flag *done* with n separate flags (one for each process) whose OR is equivalent to the original *done* bit, and performing a cheap-snapshot on the n flags after every coin flip. This guarantees that the analysis of both the total step complexity and the agreement parameter remain the same.

We remark that Aspnes [5] shows an $\Omega(\frac{n^2}{\log^2 n})$ lower bound on the expected total number of coin flips. Our layering approach as presented here, does not distinguish a deterministic step from a step involving a coin flip, leaving open the question of the amount of randomization needed.

Turning our attention to another adversarial model, in Chapter 10, we presented lower bounds for the termination probability achievable by randomized consensus algorithms with bounded run-

	Model	$k = \log n$	$k = \log^2 n$
lower bound	asynchronous MP, SWCS, MWCS	$\frac{1}{n^{\Omega(1)}}$	$\frac{1}{n^{\Omega(\log n)}}$
	synchronous MP [36]	$\frac{1}{n^{\Omega(\log \log n)}}$	
upper bound	SWMR [21]		$\frac{1}{n^{O(1)}}$
	MWMR [20]	$\frac{1}{n^{O(1)}}$	

Table 12.1: Bounds on q_k in different models, when agreement is required to always hold. MP is the message-passing model, while SW/MW stands for single/multi-writer registers, SR/MR for single/multi-reader registers, and CS for cheap snapshots.

ning time, under a very weak adversary. Our results are particulary relevant in light of the recent surge of interest in providing *Byzantine fault-tolerance* in practical, asynchronous distributed systems (e.g., [29, 55]). The adversarial behavior in these applications is better captured by non-adaptive adversaries as used in our lower bounds, rather than the adaptive adversary, which can observe the results of local coin-flips.

For all models, when agreement is required to always hold, we have shown that the probability q_k that the algorithm fails to complete in $k(n-f)$ steps is at least $\frac{1}{c^k}$, for a model-dependent value c which is a constant if $\lceil \frac{n}{f} \rceil$ is a constant. Table 12.1 shows the bounds for specific values of k.

The previous lower bound for the synchronous message-passing model [36] is $q_k \geq \frac{1}{(ck)^k}$, for some constant c. From the perspective of the expected total step complexity, given a non-termination probability δ, the lower bound of [36] implies $\Omega\left((n-f)\frac{\log 1/\delta}{\log\log 1/\delta}\right)$ steps, which is improved by our bound to $\Omega\left((n-f)\log 1/\delta\right)$ steps.

In the shared-memory model with single-writer multi-reader registers, Aumann and Bender [21] show a consensus algorithm with probability $1 - \frac{1}{n^{O(1)}}$ for terminating in $O(n \log^2 n)$ steps. For multi-writer multi-reader registers, Aumann [20] presents an iterative consensus algorithm, with constant probability to terminate at each iteration, independently of the previous iterations. This implies that the probability of terminating after k iterations is $1 - \frac{1}{c^k}$, for some constant c.

When agreement is required to hold only with high probability, Kapron et al. [52] give an algorithm for the asynchronous message passing model that always terminates within $\mathrm{polylog}(n)$ asynchronous rounds and has a probability $\frac{1}{\mathrm{polylog}(n)}$ for disagreeing, and an algorithm that always terminates within $2^{\Theta(\log^7 n)}$ asynchronous rounds and has a probability $\frac{1}{\mathrm{poly}(n)}$ for disagreeing.

An interesting open question is to close the gap between the values of the probability ϵ_k for disagreement achieved in the algorithms of [52] and the lower bounds obtained in this work on that probability. It is also interesting to tighten the bounds in the synchronous model and for large values of k.

Our lower bounds can be used to estimate the error distribution and bound the variance of the running time of randomized consensus algorithms. They do not yield significant lower bounds for the expected step complexity—there is still a large gap between the (trivial) lower bounds and upper bounds for the shared-memory model, with a weak adversary.

In addition to randomized consensus, an important topic that we address in Chapter 6 is the design of concurrent data structures. We give a method for using multi-writer multi-reader registers to construct m-bounded max registers with $\lceil \lg m \rceil$ cost per operation, and unbounded max registers with $O(\min(\log v, n))$ cost to read or write the value v. An analog data structure of a *min register* can be implemented in a similar way. In [9] we prove a lower bound that shows that the cost of our implementation is optimal. For randomized implementations, we show a lower bound of $\Omega(\log n / \log(w \log n))$ for read operations, where w is the cost of write operations. This leaves open the problem of tightening the randomized lower bound for $m \gg n$, or finding an implementation whose cost depends only on n.

We use max registers to construct wait-free concurrent data-structures out of any monotone circuit, while satisfying a natural consistency condition we call *monotone consistency*. The cost of a write is $O(Sd \min(\lceil \log m \rceil, O(n)))$, where m is the size of the alphabet for the circuit, S is the number of gates whose value changes as the result of the write, and d is the number of inputs to each gate; the cost of a read is $\min(\lceil \log m \rceil, O(n))$.

As an application, we obtain a simple, linearizable, wait-free counter implementation with a cost of $O(\min(\log n \log v, n))$ to perform an increment and $O(\min(\log v, n))$ to perform a read, where v is the current value of the counter. For polynomially-many increments, these become $O(\log^2 n)$ and $O(\log n)$, respectively, an exponential improvement on the best previously known upper bounds of $O(n)$ for an exact counting and $O(n^{4/5+\epsilon})$ for approximate counting [11]. Note that bounding the counters allows us to overcome the linear lower bound of Jayanti, Tan, and Toueg [51], as well as the similar lower bounds by Fich, Hendler, and Shavit [41] that hold even with CAS primitives. Whether further improvements are possible is still open.

Our polylogarithmic counter is used in Chapter 7 in order to obtain a randomized consensus algorithm with an optimal individual step complexity of $O(n)$.

Finally, Chapter 8 presents wait-free randomized algorithms for the set-agreement problem in asynchronous shared-memory systems. There are many open questions that arise and are interesting subjects for further research, as we elaborate next.

We extended the definition of shared-coin algorithms to multi-sided shared coins. It is an open question whether our $(k+1)$-sided shared-coin algorithm can be improved while keeping the agreement parameter constant. In addition, the definition can be modified so that the agreement parameter holds for subsets of less than k values. It is interesting to find good implementations for multi-sided shared coins that satisfy this modified definition.

For randomized set-agreement algorithms, it is open whether better algorithms exist in this model. In addition, it would be intriguing to prove lower bounds on the complexities of such algorithms, as no such bounds are currently known.

We note that for $k \leq \sqrt{n}$ the total and individual step complexities of our $(k, k+1, n)$-agreement algorithm are the same as for the optimal algorithm for randomized binary consensus, only divided by k. First, it is an open question whether the same complexities can be obtained for larger values of k. In addition, a similar relation between consensus and set agreement occurs also for complexities in deterministic synchronous algorithms and lower bounds under f failures, since the optimal number of rounds for solving consensus is $f+1$ [19], while the optimal number of rounds for solving set agreement is $f/k+1$ [33]. It is interesting whether this is a coincidence or an indication of an inherent connection between the two problems.

We believe that similar algorithms to the ones presented in this work can be constructed for weaker adversarial models, see [7] for recent work. It is an open question whether there can be improved algorithms for weaker adversaries, and it is also important to find analogous algorithms for solving set agreement in message-passing systems. Needless to say, obtaining lower bounds for these models is an important direction for further research.

Recent unpublished developments suggest that there is a randomized multi-valued consensus algorithm with the same total and individual step complexities as binary consensus, i.e., $O(n^2)$ and $O(n)$, respectively. This would improve upon the extra $\log k$ factor of the complexities of the corresponding algorithms presented here, and would be tight since by definition binary consensus

is a special case of multi-valued consensus.

As we have seen, many questions arising from the research of randomized consensus still remain open. These constitute intriguing subjects for further work. Considering other decision problems, such as *renaming*, and additional models of adversaries, are important topics as well, which lie at the heart of the theory of distributed computing.

Bibliography

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 291–302, 1988.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[3] N. Alon and J. H. Spencer. *The probabilistic Method*. John Wiley & Sons, Hoboken, New Jersey, 2nd edition, 2000.

[4] J. Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms*, 14(3):414–431, 1993.

[5] J. Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3):415–450, May 1998.

[6] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–176, Sept. 2003.

[7] J. Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 460–467, 2010.

[8] J. Aspnes, H. Attiya, and K. Censor. Randomized consensus in expected $O(n \log n)$ individual work. In *Proceedings of the 27th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 325–334, 2008.

[9] J. Aspnes, H. Attiya, and K. Censor. Max registers, counters, and monotone circuits. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 36–45, 2009.

[10] J. Aspnes, H. Attiya, and K. Censor. Combining shared-coin algorithms. *Journal of Parallel and Distributed Computing*, 70(3):317–322, 2010.

[11] J. Aspnes and K. Censor. Approximate shared-memory counting despite a strong adversary. *ACM Transactions on Algorithms*, 6(2):1–23, 2010.

[12] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, Sept. 1990.

[13] J. Aspnes and O. Waarts. Randomized consensus in expected $O(N \log^2 N)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, Oct. 1996.

[14] H. Attiya and K. Censor. Lower bounds for randomized consensus under a weak adversary. In *Proceedings of the 27th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 315–324, 2008.

[15] H. Attiya and K. Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):1–26, 2008.

[16] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 281–293, 1989.

[17] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

[18] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1st edition, 1998.

[19] H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Hoboken, New Jersey, 2nd edition, 2004.

[20] Y. Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 209–218, 1997.

[21] Y. Aumann and M. A. Bender. Efficient low-contention asynchronous consensus with the value-oblivious adversary scheduler. *Distributed Computing*, 17(3):191–207, Mar. 2005.

[22] Y. Aumann and A. Kapah-Levy. Cooperative sharing and asynchronous consensus using single-reader single-writer registers. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 61–70, 1999.

[23] Z. Bar-Joseph and M. Ben-Or. A tight lower bound for randomized synchronous consensus. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 193–199, 1998.

[24] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, 1983.

[25] M. Ben-Or, E. Pavlov, and V. Vaikuntanathan. Byzantine agreement in the full-information model in o(log n) rounds. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 179–186, 2006.

[26] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[27] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 1993.

[28] G. Bracha and O. Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG)*, pages 143–150, 1991.

[29] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[30] K. Censor Hillel. Multi-sided shared coins and randomized set-agreement. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 60–68, 2010.

[31] T. D. Chandra. Polylog randomized wait-free consensus. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 166–175, 1996.

[32] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.

[33] S. Chaudhuri, M. Herlihy, N. A. Lynch, and M. R. Tuttle. Tight bounds for k-set agreement. *Journal of the ACM*, 47(5):912–943, 2000.

[34] L. Cheung. Randomized wait-free consensus using an atomicity assumption. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS)*, pages 47–60, 2005.

[35] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 86–97, 1987.

[36] B. Chor, M. Merritt, and D. B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, 1989.

[37] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[38] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[39] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal. Randomized multivalued consensus. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Computing*, pages 195–200, 2001.

[40] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, 3rd edition, 1968.

[41] F. E. Fich, D. Hendler, and N. Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 165–173, 2005.

[42] M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

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

[44] S. Goldwasser, E. Pavlov, and V. Vaikuntanathan. Fault-tolerant distributed computing in full-information networks. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 15–26, 2006.

[45] G. R. Grimmett and D. R. Stirzaker. *Probability and Random Processes*. Oxford Science Publications, 2nd edition, 1992.

[46] P. Hall and C. Heyde. *Martingale Limit Theory and Its Application*. Academic Press, 1980.

[47] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[48] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

[49] M. Inoue, T. Masuzawa, W. Chen, and N. Tokura. Linear-time snapshot using multi-writer multi-reader registers. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, pages 130–140, 1994.

[50] P. Jayanti. f-arrays: implementation and applications. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 270–279, 2002.

[51] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[52] B. Kapron, D. Kempe, V. King, J. Saia, and V. Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1038–1047, 2008.

[53] A. Karlin and A. C.-C. Yao. Probabilistic lower bounds for byzantine agreement and clock synchronization. Unpublished manuscript.

[54] V. King, J. Saia, V. Sanwalani, and E. Vee. Scalable leader election. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*, pages 990–999, 2006.

[55] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2007.

[56] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[57] M. C. Loui and H. H. Abu-Amara. *Memory Requirements for Agreement Among Unreliable Asynchronous Processes*, pages 163–183. JAI Press, Greenwich, Conn., 1987.

[58] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[59] N. A. Lynch, R. Segala, and F. W. Vaandrager. Observing branching structure through probabilistic contexts. *SIAM Journal on Computing*, 37(4):977–1013, 2007.

[60] Y. Moses and S. Rajsbaum. A layered analysis of consensus. *SIAM Journal on Computing*, 31(4):989–1021, 2002.

[61] A. Mostefaoui and M. Raynal. Randomized k-set agreement. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 291–297, 2001.

[62] A. Mostefaoui, M. Raynal, and F. Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, 73(5-6):207–212, 2000.

[63] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambrigde University Press, Cambridge, United Kingdom, 1995.

[64] M. Raynal and M. Roy. A note on a simple equivalence between round-based synchronous and asynchronous models. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 387–392, 2005.

[65] M. Raynal and C. Travers. Synchronous set agreement: a concise guided tour (including a new algorithm and a list of open problems). In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 267–274, 2006.

[66] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus—making resilient algorithms fast in practice. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 351–362, 1991.

[67] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, 2000.

[68] N. Santoro and P. Widmayer. Time is not a healer. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 304–313, 1989.

[69] G. Schechtman. Levy type inequality for a class of finitie metric spaces. In *Lecture Notes in Mathematics: Martingle Theory in Harmonic Analysis and Banach Spaces*, volume 939, pages 211–215. Springer-Verlag, 1981.

[70] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computer Surveys*, 22(4):299–319, Dec. 1990.

[71] R. Segala. A compositional trace-based semantics for probabilistic automata. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR)*, pages 234–248, 1995.

[72] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[73] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 233–243, 1986.

[74] J. Zhang and W. Chen. Bounded cost algorithms for multivalued consensus using binary consensus instances. *Information Processing Letters*, 109(17):1005–1009, 2009.

סיבכויות מספר הצעדים האינדיוידואלית של הסכמה רנדומית תחת יריב חזק נענתה במלואה מאוחר יותר באמצעות בניית מונה מקורב תת-לינארי (כלומר עם מספר צעדים קטן אסימפטו-טית מ-n לכל פעולה על המונה). תוצאה זו מעוררת עניין במבני נתונים תת-לינאריים נוספים, ובפרט כאלו הנותנים ערכים מדויקים. אנו מציגים מונה מדויק פולילוגריתמי, המשתמש במבנה נתונים אותו אנו מכנים רגיסטר-מקסימום, שאותו אנו מממשים במספר צעדים פולילוגריתמי עבור כל פעולה. לאחר מכן, אנו מציגים מסגרת לבניית מטבע משותף בעל פרמטר הסכמה קבוע וסיבוכיות מספר צעדים אינדיוידואלית אופטימלית של $O(n)$, המשתמשת במונה מדויק פולילו-גריתמי. מטבע משותף כזה משרה אלגוריתם להסכמה רנדומית עם סיבוכיות מספר צעדים אינ-דיוידואלית אופטימלית של $O(n)$ על ידי שימוש ברדוקציה של Aspnes ו-Herlihy.

דרך אחרת לעקוף את תוצאת אי האפשרות של בעית ההסכמה היא להרשות יותר אפשרויות. ניתן לתאר גישה זו באמצעות בעית הסכמת-הקבוצה, שבה הקלטים נלקחים מקבוצה המכילה יותר משני איברים, ומותר יותר מפלט יחיד. בעיה זו הוגדרה לראשונה בעבודה של Chaudhuri, שבה גם הוכח שאפשר לפתור את בעיית הסכמת-הקבוצה בצורה דטרמיניסטית במערכת אסינכר-ונית, כל עוד מספר הנפילות קטן ממספר הערכים השונים המותרים עבור הפלטים. מאוחר יותר הוכח על ידי Borowski ו-Gafni, Herlihy ו-Shavit, ו-Saks ו-Zaharoglou, שלא קיים אלגוריתם דטרמיניסטי לפתרון בעיית הסכמת-הקבוצה במערכת אסינכרונית שבה מספר הנפילות יכול לה-גיע למספר הערכים השונים המותרים עבור הפלטים. כמו עבור בעיית ההסכמה, גם על תוצאת אי האפשרות הזו ניתן להתגבר באמצעות הוספת רנדומיזציה. בחיבור זה אנו מציגים מספר אל-גוריתמים רנדומיים עבור בעיית הסכמת-הקבוצה, העמידים בפני כל מספר של נפילות, במערכת של זכרון משותף המכילה רגיסטרים הניתנים לקריאה וכתיבה על ידי כל התהליכים. בפרט, אנו מציגים אלגוריתם להסכמת-קבוצה על k מתוך $k+1$ ערכים. לצורך כך אנו מגדירים מטבע משותף רב-צדדי, שהוא הרחבה של מטבע משותף רגיל למספר רב יותר של ערכים, ומציגים מימוש של מטבע כזה. בנוסף, אנו מראים מספר אלגוריתמים עבור פרמטרים נוספים של כמות הערכים המותרת.

התרומה העיקרית של חיבור זה היא הצגת חסם אסימפטוטי הדוק של $O(n^2)$ (כאשר n הוא מספר התהליכים) עבור סיבוכיות מספר הצעדים הכוללת של הסכמה רנדומית, שהיא תוחלת מספר הצעדים הכולל המבוצעים על ידי כל התהליכים. תוצאה מוזגת על ידי שיפור החסם העליון הקודם של $O(n^2 \log n)$ של Bracha ו- Rachman, מצד אחד, ועל ידי שיפור החסם התחתון הקודם של $\Omega(n^2 \log^2 n)$ של Aspnes, מצד שני. התוצאה מוכחת במודל של זכרון משותף, המכיל רגיסטרים הניתנים לקריאה ולכתיבה על ידי כל התהליכים. היריב השולט בתזמון הינו יריב חזק, המבסס את החלטת התזמון הבאה שלו על סמך ערכי המשתנים המשותפים והמצבים הלוקלים של כל התהליכים (כולל תוצאות הטלות המטבע).

החסם העליון מושג באמצעות הצגת אלגוריתם חדש של מטבע משותף וניתוח פרמטר ההסכמה וסיבוכיות מספר הצעדים הכוללת שלו על ידי הסתכלות על התהליך הסטכסטי שמייצג אותו. את האלגוריתם להסכמה רנדומית משיגים מהמטבע המשותף בעזרת רדוקציה של Aspnes ו- Herlihy. החסם התחתון נשען על צמצום קבוצת הריצות לריצות בעלות מבנה דמוי-סיבובים הנקראות שכבות, ושימוש בטיעוני ערכיות רנדומית על מנת למנוע מהאלגוריתם לסיים מהר מדי. אנו מראים כיצד ניתן להישאר בהסתברות גבוהה בקונפיגורציות דו-ערכיות או חסרות-ערכיות. המקרה האחרון ממודל על ידי משחקי הטלות מטבע, המנותחים בעזרת אי שויון איזופרימטרי.

בנוסף לתוצאה הנ"ל העונה במלואה על שאלת סיבוכיות מספר הצעדים הכוללת של הסכמה רנדומית תחת יריב חזק, אנו מציגים תוצאה נוספת שהיא חסם על מספר הצעדים הכולל כפונקציה של ההסתברות לסיום של אלגוריתם להסכמה רנדומית תחת יריב חלש, המחליט על כל התזמון מראש. ביתר פירוט, אנו מראים כי לכל מספר שלם k, ההסתברות שאלגורים להסכמה רנדומית העמיד בפני f נפילות לא יסיים אחרי $k(n-f)$ צעדים היא לפחות $1/c^k$, כאשר c הוא קבוע אם n/f הוא קבוע. התוצאה מתקיימת במודל של העברת הודעות, כמו גם במודל של זכרון משותף (אם כי עם קבוע שונה).

מדד נוסף אותו אנו בוחנים הוא סיבוכיות מספר הצעדים האינדיוידואלית של תהליך בודד. במ-טבעות משותפים מסורתיים תהליך בודד עלול להידרש לבצע את כל העבודה בעצמו, דבר שעודד תכנון מטבעות משותפים המצמצמים את סיבוכיות מספר הצעדים האינדיוידואלית. המחיר של שיפור זה הוא בהגדלת סיבוכיות מספר הצעדים הכוללת. בחיבור זה אנו מראים כיצד לשלב מטבעות משותפים בכדי לקבל מטבע משותף הנהנה מהטוב שבמדדי הסיבוכיות, בעוד שפרמטר ההסכמה שלו הוא מכפלת פרמטרי ההסכמה המקוריים. בעזרת האלגורים המשולב אנו משפרים תוצאות קודמות עבור הסכמה רנדומית במספר מודלים של חישוב.

עבור המודל הספציפי של רגיסטרים הניתנים לקריאה ולכתיבה על ידי כל התהליכים, שאלת

תקציר

מאפיין מהותי של מערכות מבוזרות הינו החוסר בגורם שליטה מרכזי, אשר היעדרו דורש מהרכיבים לתאם את פעולותיהם. ניתן לתאר צורך זה על ידי בעיית ההסכמה, שבה לכל תהליך יש קלט בינארי ועליו להוציא פלט בינארי, כך שכל הפלטים יהיו זהים. בעיית ההסכמה היא יסודית עבור מערכות אסינכרוניות, ומאפשרת מימוש של אוביקטים מקביליים; בנוסף, בעיית ההסכמה מהווה רכיב-מפתח בגישת מכונת-המצבים לשירותי שכפול.

אחד הקשיים בהשגת הסכמה נובע מהאפשרות לתקלות של תהליכים במערכת, כאשר נפילה של תהליך מתרחשת כאשר הוא מפסיק לבצע צעדי חישוב. אפליקציות נדרשות להיות עמידות בפני נפילות, ולהבטיח שתהליכים תקינים פועלים כדרוש. דרישה זו צריכה להתקיים ללא תלות במספר הנפילות, כלומר על האלגוריתמים להיות חסרי-המתנה.

באופן פורמלי, בעיית ההסכמה דורשת שכל תהליך תקין יתן פלט בסופו של דבר (זהו תנאי העצירה), כך שכל הפלטים יהיו זהים (זהו תנאי ההסכמה). כדי למנוע פתרונות טריויאלים, הפלט הזהה חייב להיות הקלט של אחד התהליכים (תנאי הולידיות).

איכותם של אלגוריתמים מבוזרים במערכות עם אפשרות לתקלות, ולעיתים עצם קיומם, מושפעים באופן מהותי מהנחות התזמון שמציבה המערכת. שני סוגים נפוצים של מערכות מבוזרות הן מערכות סינכרוניות ומערכות אסינכרוניות. במערכת סינכרונית החישוב מתבצע בסיבובים, שכל אחד מכיל צעד יחיד על ידי כל תהליך תקין. מהות הצעד תלויה בצורת התקשורת של המערכת. במערכת אסינכרונית, אין שום הנחות בנוגע לתזמון; אין חסם על הזמן שבין שני צעדים שונים של תהליך כלשהו, או בין שני צעדים של תהליכים שונים. לפיכך, במערכת אסינכרונית אין אפשרות להבחין בין תהליך שנפל לבין תהליך מאד איטי.

אחת התוצאות היסודיות בתיאוריה של חישוב מבוזר, הידועה כתוצאת אי האפשרות של FLP, שהוכחה לראשונה על ידי Fischer, Lynch ו- Paterson, מראה שלא קיים אלגוריתם דטרמיניסטי שפותר את בעיית ההסכמה במערכת אסינכרונית, אף אם מובטח שלא נופל יותר מתהליך בודד.

כיוון שבעיית ההסכמה הינה אבן פינה בחישוב מבוזר, מחקר רב בוצע על מנת להתגבר על תוצאת אי האפשרות הזו. אחת הגישות המאפשרות הסכמה במערכת אסינכרונית עם תקלות הינה הוספת רנדומיזציה לחישוב. בגישה זו ניתנת האפשרות לתהליכים לסיים בהסתברות 1 במקום בכל ריצה אפשרית, בעוד שתנאיי ההסכמה והולידיות נדרשים בכל ריצה.

הסכמה רנדומית הינה הנושא המרכזי בו עוסק חיבור זה, החוקר אלגוריתמים וחסמים תחתונים לבעיה. בנוסף, החיבור עוסק בבעיות הנובעות ממחקר הסכמה רנדומית, כולל הסכמת-קבוצה, ומבני נתונים מקביליים יעילים.

תודות

המחקר נעשה בהנחיית פרופסור חגית עטיה בפקולטה למדעי המחשב.

אני אסירת תודה לחגית על היותה הרבה מעבר למנחה לדוקטורט. ההורות האקדמית שלה הכינה אותי, בכל כך הרבה מובנים, לקראת המשך מסעי בעולם המחקר. אני חשה ברת מזל על כך שהיתה המנחה שלי, ולעולם לא אוכל להודות לה די עבור סבלנות ותוכמה אין קץ.

אני מודה לפרופ' ג'יימס אספנס מאוניברסיטת ייל ולפרופ' הדס שכנאי מהטכניון על שיתופי פעולה פוריים. היה לי העונג לעבוד עם שניהם, ולמדתי מהם רבות.

פרופ' יהודה אפק מאוניברסיטת תל-אביב ופרופ' יורם מוזס מהטכניון נתנו לי הערות והצעות מועילות על גרסא קודמת של חיבור זה (ובכלל), ועל כך אני מודה לשניהם.

חברים רבים בטכניון עזרו להפוך את השנים האלו לתקופה שלא תישכח. אני מודה לדוד חי, יניב כרמלי, חנא מזאווי, רועי רונן, תמר אייזיקוביץ', סיוון ברקוביץ', וסוניה ליברמן, על ידידות אמיצה, הרבה מעבר לאינספור הפסקות הקפה המשותפות.

לבסוף, אני מודה להוריי, ערגה ויאיר, ולדודתי ודודי, איה וצבי, על אהבתם ותמיכתם. ולגיל, על היותו בֵּיתי. חיבור זה מוקדש להם.

אני מודה לטכניון, לקרנות נאמן וג'ייקובס-קואלקום, ולתוכנית מלגות אדאמס של האקדמיה הלאומית למדעים, על התמיכה הכספית הנדיבה בהשתלמותי.

שיטות הסתברותיות בחישוב מבוזר

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת תואר
דוקטור לפילוסופיה

קרן צנזור הלל

הוגש לסנט הטכניון - מכון טכנולוגי לישראל

אלול תש"ע חיפה אוגוסט 2010

שיטות הסתברותיות בחישוב מבוזר

קרן צנזור הלל