On the Complexity of Synchronization

by

Rati Gelashvili

B.S., Swiss Federal Institute of Technology (2012)
S.M., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2017

Certified by . . . .
Nir Shavit
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

On the Complexity of Synchronization

by

Rati Gelashvili

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2017, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

The field of distributed algorithms revolves around efficiently solving *synchronization tasks*, such as *leader election* and *consensus*. We make contributions towards a better understanding of the complexity of central tasks in standard distributed models.

In the population protocols model, we demonstrate how to solve *majority* and *leader election* efficiently, in time $O(\log^2 n)$, using $O(\log n)$ states per node, for n nodes. Central to our algorithms is a new *leaderless phase clock* technique. We also prove tight lower bounds on the state complexity of solving these tasks.

In shared memory, we prove that any nondeterministic solo terminating consensus algorithm for anonymous processes has to use $\Omega(n)$ read-write registers. Then, we show how to solve n-process wait-free consensus by combining synchronization instructions that would be considered "weak" according to Herlihy's consensus hierarchy. This collapses the hierarchy when instructions can be applied to the same memory location, as is the case in all existing multicore processors. We suggest an alternative hierarchy and provide a practical universal construction using only "weak" instructions, that performs as well as the *Compare-and-Swap*-based solution.

Space complexity of solving k-set agreement is a problem that highlights important gaps in our understanding and state-of-the-art methods. No general lower bound better than 2 is known. We introduce a new technique based on an indirect black-box application of Sperner's Lemma through an algorithmic reduction to the impossibility of wait-free k-set agreement. We design a simulation such that for any protocol Π either the simulating processes solve wait-free k-set agreement (impossible), or they simulate an execution of Π that uses many registers.

Finally, time complexity of *leader election* is a long-standing open problem. We give an algorithm with $O(\log^\star k)$ time complexity in asynchronous message-passing system, for k participants.

Thesis Supervisor: Nir Shavit

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor Nir Shavit for supervision, an incredible support and for introducing me to the exciting field of concurrent algorithms. It is my privilege and pleasure to be a student of Nir who I respect and admire as a wonderful human being and a brilliant researcher and advisor, my true role model. In many ways interaction with Nir shaped me as a scientist, a friend and a world citizen.

I want to thank Professor Faith Ellen - for thorough feedback, outstanding support and encouragement. I am very much looking forward to the postdoc under her supervision.

Special thanks to Professors James Aspnes and Aleksander Madry for their valuable feedback and the very fact of serving on my thesis committee.

This thesis is based on joints works with Dan Alistarh, Jim Aspnes, David Eisenstat, Faith Ellen, Idit Keidar, Ron Rivest, Nir Shavit, Alexander (Sasha) Spiegelman, Adrian Vladu, Milan Vojnovic, Roger Wattenhofer, Leqi (Jimmy) Zhu. Razor-sharp discussions were part of these enjoyable intellectual journeys under which I learnt, grew, and matured as a scientist.

I would like to specially thank Dan Alistarh for taking me under his wing as a young student, and Jimmy Zhu for the most productive, challenging and very pleasant collaborations. I would also like to thank Philipp Woelfel, Yoram Moses, Idit Keidar, Sasha Spiegelman, Faith Ellen, Jimmy Zhu, Dan Alistarh for hosting me as a visitor at various points during my PhD - you have been amazing hosts and inspiring presence!

Family and friends from all around the world - without your support I would not have been where I am today!

Friends in the Boston area with whom I spent past five years (including frosty winters). Among them I want to say special thanks to Guranda Darchidze, Ilya Razenshteyn, Shibani Santurkar, David Budden, Adrian Vladu, Mohsen Ghaffari, Jerry Li, Guatam [1] "G" Kamath, Mira Radeva, Mari Kobiashvili, Tornike Metreveli, Sasha Konstantinov, Achuta Kadambi, Merav Parter and Stephan Holzer.

[1]Gautam

A huge part of this experience has been internships. Thus, I would like to thank Milan Vojnovic, Dan Alistarh, David Amenhauser, Debabrata Banerjee, Bogdan Munteanu, Diwaker Gupta, Mirjam Wattenhofer and Kriti Puniyani for providing outstanding personal and work environment.

I am very thankful to MIT Department of Computer Science and Electrical Engineering for providing the funding for my education and research.

Lastly, I want to mention Soso (Sensei) Dzmanashvili for unconditional support and extracurricular inspiration.

Contents

1 Introduction **13**

1.1 On Population Protocols 14

1.1.1 Leader Election and Majority 16

1.1.2 Summary 19

1.1.3 Chapter Outline 19

1.2 On Shared Memory 22

1.2.1 Complexity-Based Hierarchy 22

1.2.2 Towards Reduced Instruction Sets for Synchronization 24

1.2.3 Anonymous Space Lower Bound 25

1.2.4 *k*-Set Agreement 27

1.2.5 Chapter Outline 30

1.3 On Message Passing 32

2 Population Protocols **35**

2.1 Model 35

2.2 Leader-Minion Algorithm 40

2.2.1 Analysis 42

2.3 Leaderless Phase Clock 48

2.3.1 Analysis 49

2.4 Phased Majority Algorithm 51

2.4.1 Analysis 55

2.5 Synthetic Coin Flips 66

2.5.1 Analysis 66

2.5.2 Approximate Counting 69
2.6 Phased Leader Election 69
2.6.1 Analysis 71
2.7 Lower Bounds 72
2.7.1 Technical Tools 72
2.7.2 Output-Dominant Majority 78
2.7.3 General Lower Bound 84

3 Shared Memory **97**
3.1 Anonymous Space Lower Bound 97
3.1.1 Definitions and Notation 98
3.1.2 A Square-Root Lower Bound 100
3.1.3 Linear Lower Bound 104
3.2 The Space Hierarchy 117
3.2.1 Model 117
3.2.2 Arithmetic Instructions 118
3.2.3 Increment 121
3.2.4 Buffers 123
3.2.5 Multiple Assignment 132
3.3 Universality using "Weak" Instructions 140
3.3.1 Algorithm 142
3.4 Enter the Simulation: k-Set Agreement 145
3.4.1 Model 145
3.4.2 Local Argument 146
3.4.3 Global Argument 151

4 Message Passing **163**
4.1 Definitions and Notation 163
4.2 Leader Election Algorithm 166
4.2.1 The PoisonPill Technique 166
4.2.2 Heterogeneous PoisonPill 167

4.2.3 Final construction 170

5 Conclusions **179**

List of Figures

1-1 Summary of results on Majority and Leader Election. 20

2-1 The state update rules for the *LM* algorithm. 41

2-2 Pseudocode for the phased majority algorithm, part 1/2 95

2-3 Pseudocode for the phased majority algorithm, part 2/2 96

3-1 Proof of Lemma 3.1.10, Case 1 112

3-2 Proof of Lemma 3.1.10, Case 2 114

3-3 Illustration of Case 2 in History object emulation 126

3-4 Element of A. 142

3-5 Pseudocode for the Implementation of a 1-Augmented Snapshot Object 154

4-1 PoisonPill Technique 166

4-2 Heterogeneous PoisonPill 168

4-3 PreRound procedure 171

4-4 Doorway procedure 172

4-5 Leader election algorithm 172

Chapter 1

Introduction

To solve a problem in a distributed fashion, nodes peforming parts of the computation need to synchronize with each other. The common synchronization requirements are often abstracted and captured by *synchronization tasks*. Examples of important synchronization tasks include *consensus (agreement)* [LSP82, PSL80], *leader election (test-and-set)* [AGTV92], *majority* [Tho79], *mutual exclusion* [Dij65], *renaming* [ABND$^+$90], *task allocation (do-all)* [KS92] and *timestamps* [Lam78].

These tasks are usually considered in two classical models for distributed computation: *asynchronous shared memory* and *asynchronous message-passing* [Lyn96]. Additionally, population protocols [AAD$^+$06] are a popular model of distributed computing, in which randomly-interacting agents with little computational power cooperate to jointly perform complex computation.

In all these models, randomization plays a critical role in solving synchronization tasks. In population protocols, interactions happen according to a randomized scheduler. In the classical models, celebrated impossibility results of [FLP85, HS99] limit the power of deterministic distributed computation. Fortunately, relaxing the task specifications to allow for randomization [BO83] (and in particular, probabilistic termination) has proved a very useful tool for circumventing fundamental impossibilities, and for obtaining efficient algorithms.

In each model, there are two standard complexity measures. *Time complexity*, defined appropriately, can be formulated in all three models. In population protocols, the other (more important) measure is *state complexity* of the agents. In asynchronous shared memory, *space complexity* denotes the number of shared memory locations that can be concurrently accessed by processors. Such locations have historically also been called *registers*. In asynchronous message-passing, the other natural measure is *message complexity* of an algorithm.

The thesis is divided into three main chapters, each presenting results related to one model. We describe the results in more detail below.

1.1 On Population Protocols

Population protocols [AAD$^+$06] are a model of distributed computing in which agents with very little computational power and interacting randomly cooperate to collectively perform computational tasks. Initially introduced to model animal populations [AAD$^+$06], they have proved a useful abstraction for settings from wireless sensor networks [PVV09, DV12], to gene regulatory networks [BB04], and chemical reaction networks [CCDS15]. In this last context, there is an intriguing line of applied research showing that population protocols can be implemented at the level of DNA molecules [CDS$^+$13], and that some natural protocols are equivalent to computational tasks solved by living cells in order to function correctly [CCN12].

A population protocol consists of a set of n finite-state agents, interacting in randomly chosen pairs, where each interaction may update the local state of both participants. A *configuration* captures the "global state" of the system at any given time: since agents are anonymous, the configuration can be entirely described by the number of nodes in each state. The protocol starts in some valid initial configuration, and defines the outcomes of pairwise interactions. The goal is to have all agents stabilize to some configuration, representing the output of the computation, such that all future configurations satisfy some predicate over the initial configuration of the system.

In the fundamental *majority* task [AAE08b, PVV09, DV12], agents start in one of two input states A and B, and must stabilize on a decision as to which state has a higher initial count. Another important task is *leader election* [AAE08a, AG15, DS15], which requires the system to stabilize to final configurations in which a *single* agent is in a special *leader* state.

We work in a standard setting where, at each step, a *probabilistic* scheduler picks the next pair of nodes to interact uniformly at random among all pairs. One key complexity measure for algorithms is *parallel time*, defined as the number of pairwise interactions until stabilization, divided by n, the number of agents. The other is *state complexity*, defined as the number of *distinct states* that an agent can internally represent.

We focus on *exact* tasks, in which the protocol must return the correct decision in all executions, as opposed to *approximate* tasks, such as approximate majority [AAE08b], where the system may stabilize to wrong output configurations with low probability. Moveover, we consider parallel time until *stabilization*, i.e. until the first moment after which the configuration is guaranteed to always satisfy the correct output requirements (regardless of subsequent interactions), versus *convergence*, which is the actual moment after which the configuration always satisfies the output requirements, despite possibly non-zero probability of further divergence. Stabilization may not occur earlier than convergence. Thus, our algorithms have the same guarantees on convergence, but our lower bounds only apply to parallel time to stabilization.

Evidence suggests that the cell cycle switch in eukaryotic cells solves an approximate version of majority [CCN12]; a three-state population protocol for approximate majority was empirically studied as a model of epigenetic cell memory by nucleosome modification [DMST07]; Also, both majority and leader election are key components when simulating register machines via population protocols [AAD$^+$06, AAE08a, AAE08b]. Moreover, known efficient constructions for computing certain predicates rely on existence of a node in a leader state [AAE08a, CDS14]. Thus, it is not surprising that there has been a considerable interest in the complexity of these tasks [AAE08b, AAE08a, PVV09, DV12, CCN12, DS15, AG15, BFK$^+$16, AAE$^+$17].

1.1.1 Leader Election and Majority

A progression of deep technical results [Dot14, CCDS15] culminated in Doty and Soloveichik showing that *leader election in sublinear time is impossible* for protocols which are restricted to a *constant* number of states per node [DS15].

At the same time, we designed a simple algorithm called "Leader-Minion", that solves leader election in $O(\log^3 n)$ parallel time and requires $O(\log^3 n)$ states per node [AG15]. The algorithm follows basic and common convention that every agent starts as a potential leader, and whenever two leaders interact, one drops out of contention. Once only a constant number of potential leaders remain, they take a long time to interact, implying super-linear stabilization time. To overcome this problem, we introduce a propagation mechanism, by which contenders compete by comparing their seeding, and the nodes who drop out of contention become "minions" and assume the identity of their victor, causing nodes still in contention but with lower seeding to drop out.

In [AAE$^+$17], we gave a new leader election algorithm called "Lottery-Election", which uses $O(\log^2 n)$ states, and stabilizes in $O(\log^{5.3} n \log\log n)$ parallel time in expectation and $O(\log^{6.3} n)$ parallel time with high probability. This reduces the state space size by a logarithmic factor at the cost of a poly-logarithmic running time increase over [AG15]. We achieve this by introducing a new *synthetic coin* technique, which allows nodes to generate almost-uniform local coins within a *constant* number of interactions, by exploiting the randomness in the scheduler, and in particular the properties of random walks on the hypercube. Synthetic coins can be used to estimate the total number of agents in the system, and may be of independent interest as a way of generating randomness in a constrained setting. We employ synthetic coins to "seed" potential leaders randomly, which lets us reduce the number of leaders at an accelerated rate compared to [AG15]. This in turn reduces the maximum seeding that needs to be encoded, and hence the number of states required by the algorithm.

In [AAE$^+$17], we also improved the lower bound of [DS15]. We show that there exist constants $c \in (0,1)$ and $K \geq 1$, such that any protocol using $\lambda_n \leq c \log\log n$

states and electing at most $\ell(n)$ leaders, requires $\Omega(n/(K^{\lambda_n} \cdot \ell(n)^2))$ expected time to stabilize. Specifically, any protocol electing polylogarithmically many leaders using $\leq c \log\log n$ states requires $\Omega(n/\text{polylog } n)$ time[1].

The lower bound argument of [AAE+17] provides a unified analysis: the bounds for leader election and majority are corollaries of the main theorem characterizing the existence of certain "stable" configurations. When applied to majority, our lower bound shows that there exist constants $c \in (0,1)$ and $K \geq 1$ such that any protocol using $\lambda_n \leq c \log\log n$ states must take $\Omega(n/(K^{\lambda_n} + \epsilon n)^2))$ time to stabilize, where ϵn is the initial discrepancy between the counts of the two input states. For example, any protocol using $\leq c \log\log n$ states, even if the initial discrepancy is polylogarithmic in n, takes $\Omega(n/\text{polylog } n)$ parallel time to stabilize. The only prior lower bound was proved by us in [AGV15b], showing that sublinear time is impossible using at most *four* states per node.

In [AGV15b], we designed a *poly-logarithmic time* protocol called "AVC" which requires a number of states per node that is *linear* in n. The AVC algorithm stabilizes in poly-logarithmic time using poly-logarithmic states under a restricted set of initial configurations, e.g. assuming that the discrepancy ϵn between the two input states is large. In [AAE+17], we gave a new poly-logarithmic-time algorithm for majority, called "Split-Join", that runs in $O(\log^3 n)$ time both in expectation and with high probability, and uses $O(\log^2 n)$ states per node. This improves on [AGV15b, BFK+16] that require at least polinomially many states in n per node for achieving polylogarithmic stabilization time. Morover, the time-space bounds for the Split-Join algorithm hold for worst-case initial configurations, i.e. for $\epsilon = 1/n$. The idea is encode output opinions and their relative strength as *integer values*: a node with positive (or negative) value supports a majority of A (or of B, respectively). A higher absolute value means higher "confidence" in the corresponding output. Whenever two nodes meet, they *average* their values. This is the template used in [AGV15b], but to reduce the

[1]It is interesting to note that by Chatzigiannakis et al. [CMN+11] identified $\Theta(\log\log n)$ as a state complexity threshold in terms of the *computational power* of population protocols, i.e. the set of predicates that such algorithms can compute. Their results show that variants of such systems in which nodes are limited to $o(\log\log n)$ space per node are limited to only computing *semilinear predicates*, whereas $O(\log n)$ space is sufficient to compute general symmetric predicates.

state space further in the Split-Join algorithm, we rely on a new *quantized* averaging technique in which nodes represent only certain values as output estimates. Recently, reference [BCER17] gave a different protocol using $O(\log^2 n)$ states, but with a better stabilization time of $O(\log^2 n)$.

Recently, in [AAG17], we introduced a new synchronization construct, called a *leaderless phase clock*. A phase clock is a gadget which allows nodes to have an (approximate) common notion of time, by which they collectively count time in phases of $\Theta(n \log n)$ interactions, with bounded skew. The phase clock ensures that all nodes will be in the same phase during at least $\Theta(\log n)$ interactions of each node. Phase clocks are critical components of generic register simulations for population protocols, e.g. [AAER07]. However, they are rarely used for algorithm design, since all known constructions require the existence of a *unique leader*, which is expensive to generate. One key innovation behind our algorithm is that it is *leaderless*, as nodes maintain the shared clock collectively, without relying on a special leader node. At the implementation level, the phase clock is based on a simple but novel connection to load balancing by power of two choices, e.g. [KLMadH92, ABKU99, BCSV06, PTW15].

We build on the phase clock to obtain a new space-optimal algorithm for majority, called Phased-Majority. In a nutshell, the algorithm splits nodes into *workers*, whose job is to compute the majority value, and *clocks*, which implement a leaderless phase clock, and maintains a proportion between the counts of nodes of these types. Splitting a state space in different types is common, i.e. in the Leader-Minion algorithm, where each state is either a leader or a minion. However, doing this explicitly at the beginning of the protocol and maintaining a proportion of the counts is a recent algorithmic idea due to Ghaffari and Parter [GP16]. Workers alternate carefully-designed *cancellation* and *doubling* phases, inspired by a similar mechanism in [AAE08a]. In the former phases, nodes of disagreeing opinions as to the initial majority cancel each other out, while the latter, nodes attempt to spread their current opinion. These dynamics ensure stabilization in expected $O(\log n \cdot \log \frac{1}{\epsilon})$ parallel time, both in expectation and with high probability.

We further exploit the phase clock to obtain a simple phased algorithm for leader election. Here, the nodes are split into *contenders*, whose job is to elect a leader within themselves, and *clocks*, which implement a phase clock. Remarkably, to design an efficient algorithm, we still need to combine this with the idea of having minions (that we call *followers* in this algorithm), and use synthetic coin flips to break ties between contenders. The resulting algorithm uses $O(\log n)$ states, and stabilizes in $O(\log^2 n)$ parallel time, both in expectation and with high probability. Based on a different phase clock construction, an independent parallel work by Gąsieniec and Stachowiak [GS17] has designed a leader election protocol using $O(\log\log n)$ states. This is optimal due to the unified lower bound of [AAE+17] for majority and leader election. Combined, our results and [GS17] demonstrate an interesting separation between the state complexities of these tasks.

1.1.2 Summary

The results described above and summarized in Figure 1-1 highlight trade-offs between the running time of a population protocol and the number of states, available at each agent. The premise of the population protocols model has always been the simplicity of the agents. In its applications, it is also imperative that the agents have as low state complexity as possible. For instance, technical constraints limit the number of states currently implementable in a molecule [CDS+13]. One such technical constraint is the possibility of *leaks*, i.e. spurious creation of states following an interaction [TWS15]. In DNA implementations, the more states a protocol implements, the higher the likelihood of a leak, and the higher the probability of divergence. Time efficiency is also critical in practical implementations. Since n, the number of agents is usually quite large, it is standard to require that the parallel time until stabilization must be at most polylogarithmic in n.

1.1.3 Chapter Outline

To exemplify the model, we will start the chapter by describing our Leader-Minion algorithm and proving its state and time complexity of $O(\log^3 n)$. Then, we will focus

Problem	Expected Time Bound	Number of States	Reference
Exact Majority $\epsilon = 1/n$	$O(n \log n)$	4	[DV12, MNRS14]
	$O(\log^2 n)$	$\Theta(n)$	[AGV15b]
	$O(\log^3 n)$	$O(\log^2 n)$	[AAE$^+$17]
	$O(\log^2 n)$	$O(\log^2 n)$	[BCER17]
	$O(\log 1/\epsilon \cdot \log n)$	$O(\log n)$	[AAG17]
	$\Omega(n)$	≤ 4	[AGV15b]
	$\Omega(\log n)$	any	[AGV15b]
	$\Omega(n/\mathsf{polylog} n)$	$< 1/2 \log \log n$	[AAE$^+$17]
	$O(n^{1-c}), c > 0$	$\Omega(\log n)$	[AAG17]
Leader Election	$O(\log^3 n)$	$O(\log^3 n)$	[AG15]
	$O(\log^{5.3} n \log \log n)$	$O(\log^2 n)$	[AAE$^+$17]
	$O(\log^2 n)$	$O(\log n)$	[AAG17]
	$O(\log^2 n)$	$O(\log \log n)$	[GS17]
	$\Omega(n)$	$O(1)$	[DS15]
	$\Omega(n/\mathsf{polylog} n)$	$< 1/2 \log \log n$	[AAE$^+$17]

Figure 1-1: Summary of results on Majority and Leader Election.

on developing tools that enable better algorithms. We will describe our the leaderless phase clock construction from [AAG17] and the phased majority algorithm. Next, we will explain and prove the synthetic coin technique from [AAE$^+$17] that extracts the randomness from the scheduler and allows us to simulate almost fair coin flips, followed by the phased leader election algorithm that uses both the phase clock and synthetic coins.

Then, we will describe the lower bounds from [AAE$^+$17]. At a high level, the results of [DS15, AAE$^+$17] employ three technical steps. The first step proves that, from an initial configuration, every algorithm must reach a *dense* configuration, where all states that are expressible by the algorithm are present in large count. The second step consists of applying a *transition ordering lemma* of [CCDS15] which establishes properties that the state transitions must have in order reduce certain state counts fast from dense configurations. In the third step, these properties are used to perform careful ad-hoc *surgery* arguments to show that any algorithm that stabilizes to a correct output faster than allowed using few states must necessarily reach “stable” configurations[2] in which certain low-count states can be “erased,” i.e., may disappear

[2]Roughly, a configuration is stable if it may not generate any new types of states.

completely following a sequence of interactions. This implies executions in which the algorithm stabilizes to the wrong output, for example by engineering examples where these low-count states are exactly the set of all possible leaders.

One difference between [AAE$^+$17] and [DS15] is a stronger version of the main density theorem of [Dot14] used in [DS15] for dense configurations. Another key difference is that in [AAE$^+$17], we develop a new characterization of stable configurations, without requiring constant bounds on state space size. This also necessitates non-trivial modifications to the surgery sequences that erase the required states.

A fundamental barrier to better lower bounds is that the *first step* does not hold for algorithms using, e.g. $O(\sqrt{\log n})$ states. With such a state space, is possible to build algorithms which never go through a configuration where all states are expressed in high counts. In this thesis, we will also prove a lower bound from [AAG17] that circumvents this challenge, showing that any algorithm for majority which stabilizes in expected time $O(n^{1-c})$ for $c > 0$ requires $\Omega(\log n)$ states. We develop a generalization of the transition ordering lemma, and a new general surgery technique, which do not require the existence of dense configurations. This lower bound requires an additional assumption that we call *output dominance*, which we will discuss in detail in Chapter 2 and which is satisfied by all existing majority algorithms. Since we eliminate the density requirement, our lower bound technique applies even if the algorithm only stabilizes fast when initial configuration is equipped with a leader, which is a significant generalization over previous arguments. It can also be generalized to other types of predicates, such as equality.

We should note that [AAE08a] provides a protocol using a constant number of states and with a polylogarithmic parallel convergence time if the initial configuration is equipped with a leader. Our lower bound applies to such initial configurations and demonstrates an interesting separation, as for similarly fast stabilization, $\Omega(\log n)$ states would be necessary.

1.2 On Shared Memory

1.2.1 Complexity-Based Hierarchy

Herlihy's Consensus Hierarchy [Her91] assigns a consensus number to each object, namely, the number of processes for which there is a wait-free binary consensus algorithm using only instances of this object and read-write registers. It is simple, elegant and, for many years, has been our best explanation of synchronization power.

Robustness says that, using combinations of objects with consensus numbers at most k, it is not possible to solve wait-free consensus for more than k processes [Jay93]. The implication is that modern machines need to provide objects with infinite consensus number. Otherwise, they will not be universal, that is, they cannot be used to implement all objects or solve all tasks in a wait-free (or non-blocking) manner for any number of processes [Her91, Tau06, Ray12, HS12]. Although there are ingenious non-deterministic constructions that prove that Herlihy's Consensus Hierarchy is not robust [Sch97, LH00], it is known to be robust for deterministic one-shot objects [HR00] and deterministic read-modify-write and readable objects [Rup00]. It is unknown whether it is robust for general deterministic objects.

In adopting this explanation of computational power, we failed to notice an important fact: multiprocessors do not compute using synchronization objects. Rather, they apply synchronization instructions to locations in memory. With this point of view, Herlihy's Consensus Hierarchy no longer captures the phenomena we are trying to explain.

For example, consider two simple instructions:

- *fetch-and-add*(2), which returns the number stored in a memory location and increases its value by 2, and
- *test-and-set*(), which returns the number stored in a memory location and sets it to 1 if it contained 0[3].

[3]This definition of *test-and-set* is slightly stronger than the standard one, which always sets the location to 1. However, they have the same consensus number and they behave identically if the values are always binary.

Objects that support only one of these instructions as an operation have consensus number 2 and cannot be combined to solve wait-free consensus for 3 or more processes. However, in a system that supports both instructions, it is possible to solve wait-free binary consensus for any number of processes. The protocol uses a single memory location initialized to 0. Processes with input 0 perform *fetch-and-add*(2), while processes with input 1 perform *test-and-set*(). If the value returned is odd, the process decides 1. If the value 0 was returned from *test-and-set*(), the process also decides 1. Otherwise, the process decides 0.

Another example considers three instructions:

- *read*(), which returns the number stored in a memory location,
- *decrement*(), which decrements the number stored in a memory location and returns nothing, and
- *multiply*(x), which multiplies the number stored in a memory location by x and returns nothing.

A similar situation arises: Objects that support only two of these instructions have consensus number 1 and cannot be combined to solve wait-free consensus for 2 or more processes. However, in a system that supports all three instructions, it is possible to solve wait-free binary consensus for any number of processes. The protocol uses a single memory location initialized to 1. Processes with input 0 perform *decrement*(), while processes with input 1 perform *multiply*(n). The second operation by each process is *read*(). If the value returned is positive, then the process decides 1. If it is negative, then the process decides 0.

For randomized computation, Herlihy's Consensus Hierarchy also collapses: randomized wait-free binary consensus among any number of processes can be solved using only read-write registers, which have consensus number 1. Fich, Herlihy, and Shavit [FHS98] proved that $\Omega(\sqrt{n})$ historyless objects, which support only trivial operations, such as *read*, and historyless operations, such as *write*, *test-and-set*, and *swap*, are necessary to solve this problem. They noted that, in contrast, one fetch-and-increment or fetch-and-add object suffices for solving this problem. Yet, these objects and historyless objects are similarly classified in Herlihy's Hierarchy (i.e. they all have consensus number 1 or 2). They suggested that the number of instances of an object needed to solve randomized wait-free consensus among n processes might be another way to classifying its power.

Based on these observations, we consider a classification of instruction sets based on the number of memory locations needed to solve *obstruction-free n-valued* consensus among $n \geq 2$ processes. Obstruction freedom [HLM03] is a simple and natural progress measure. Some state-of-the-art synchronization operations, for example hardware transactions [Int12], do not guarantee more than obstruction freedom. Obstruction freedom is also closely related to randomized computation. In fact, any (deterministic) obstruction free algorithm can be transformed into a randomized wait-free algorithm that uses the same number of memory locations (against an oblivious adversary) [GHHW13]. Obstruction-free algorithms can also be transformed into wait-free algorithms in the unknown-bound semi-synchronous model [FLMS05].

1.2.2 Towards Reduced Instruction Sets for Synchronization

Contrary to common belief, our work [EGSZ16] described above has shown that computability does not require multicore architectures to support "strong" synchronization instructions like *compare-and-swap*, as opposed to combinations of "weaker" instructions like *decrement* and *multiply*. However, this is the status quo, and in turn, most efficient concurrent data-structures heavily rely on *compare-and-swap* (e.g. for swinging pointers and in general, conflict resolution).

In [GKSW17], we show that this need not be the case, by designing and implementing a concurrent linearizable Log data-structure (also known as a History object), supporting two operations: *append*(*item*), which appends the item to the log, and *get-log*(), which returns the appended items so far, in order. Readers are wait-free and writers are lock-free, and this data-structure can be used in a lock-free universal construction to implement any concurrent object with a given sequential specification. Our implementation uses atomic *read*, *xor*, *decrement*, and *fetch-and-increment* instructions supported on X86 architectures, and provides similar performance to a *compare-and-swap*-based solution on today's hardware. This raises a fundamental question about minimal set of synchronization instructions that the architectures have to support.

1.2.3 Anonymous Space Lower Bound

The above considerations motivated us to investigate the space complexity of solving obstruction-free and randomized wait-free consensus in a system in which processes communicate using only read-write registers, which was a long-standing open question. The space complexity of such an algorithm is defined as the maximum number of registers used in any execution. A lot of research has been dedicated to improving space complexity upper and lower bounds for canonical tasks.

For instance, for test-and-set, an $\Omega(\log n)$ lower bound was shown in [SP89]. On the other hand, an $O(\sqrt{n})$ deterministic obstruction-free upper bound was given in [GHHW13]. The final breakthrough was the recent obstruction-free algorithm designed by Giakkoupis et al. [GHHW15], with $O(\log n)$ space complexity. For timestamps, an implementation due to Lamport [Lam74] uses n single-writer registers. Later, in [EFR08], Ellen, Fatourou and Ruppert gave an algorithm using $n-1$ registers and proved a $\frac{1}{2}\sqrt{n-1}$ lower bound on the number of registers required to implement T. The lower bound was improved to $n/6-1$ by Helmi, Higham, Pacheco, and Woelfel in [HHPW14].

For consensus, randomized wait-free algorithms that work against strong adversary and use $O(n)$ read-write registers are long known [Abr88, AH90, SSW91, Zhu15]. Algorithms that solve consensus in a deterministic obstruction-free manner using $O(n)$ registers are also known [GR05, Bow11, Zhu15, BRS15]. A lower bound of $\Omega(\sqrt{n})$ by Fich et al. [FHS98] first appeared in 1993. The proof is notorious for its technicality and utilizes a neat inductive combination of covering and valency arguments. Another version of the proof appeared in a textbook [AE14b]. The authors of [FHS98] conjectured a tight lower bound of $\Omega(n)$, but such a bound or a sublinear space algorithm remained elusive up until very recently, when Zhu showed a lower bound of $n-1$ registers [Zhu16].

For the intervening two decades, however, the linear lower bound had not been proven even in the restricted case, when the processes are anonymous [AGM02, FHS98]. In such a system, processes have no identifiers and can be thought of as running the same code: all processes with the same input start in the same initial state, behave identically and remain in the same states until they read different values or observe different coin flip outcomes.

The linear upper bound holds for anonymous processes, as a number of deterministic obstruction-free and randomized wait-free (against strong adversary) consensus algorithms that use $O(n)$ registers are anonymous [GR05, Zhu15, BRS15]. In the algorithms of [Zhu15], the processes are memoryless (do not use local memory), in addition to being anonymous, and exactly n registers are used. In this further restricted model, [Zhu15] also showed that n registers are actually necessary.

The authors of [FHS98] introduced the notion of *clones* of processes, which has since become a standard tool for proving anonymous space lower bounds. They first showed the $\Omega(\sqrt{n})$ lower bound for anonymous processes, and then extended it to a much more complex argument for the general case. Our paper [Gel15] preceded the result of [Zhu16] and showed an $\Omega(n)$ lower bound in the anonymous case for consensus algorithms satisfying the standard *nondeterministic solo termination* property. Any lower bound for algorithms satisfying the nondeterministic solo termination implies a lower bound for deterministic obstruction-free and randomized wait-free algorithms. As in [FHS98, AE14b], the bound is for the worst-case space complexity of the algorithm, i.e. for the number of registers used in some execution, regardless of its actual probability.

Our argument relies heavily on the anonimity of the processes, and introduces specific techniques that we hope will be useful for future work. We design a class of executions, which we call *reserving*, and define a notion of valency which corresponds to the possible return values for these executions. We also extend the role of the *clones* of processes, by considering pairs of processes that can be split and reunited. This enables proving larger lower bounds by reusing the clones.

1.2.4 k-Set Agreement

The k-set agreement problem is a classical synchronization task, introduced by Chaudhuri [Cha93], where n processes, each with an input value, are required to return at most k different input values in any execution. This is a generalization of the fundamental consensus task, which corresponds to the setting when $k = 1$.

Some of the most celebrated results in the field of distributed algorithms are the impossibility of solving consensus deterministically when at most one process may crash [FLP85] and, more generally, the impossibility of solving k-set agreement deterministically when at most k processes may crash [BG93, HS99, SZ00], using only read-write registers. One way to bypass these impossibility results is to design algorithms that are *obstruction-free* [HLM03]. Obstruction-freedom is a termination condition that only requires a process to return in its *solo executions*, i.e. if a process takes sufficiently many steps on its own. x-obstruction-freedom [Tau17] generalizes this condition: in any sufficiently long execution where at most x processes take steps, these processes are required to return. It is known that k-set agreement can be solved using only registers in an x-obstruction-free way for $x \leq k$ [YNG98]. Another way to overcome the impossibility of solving consensus is using randomized wait-free algorithms, where non-faulty processes are required to terminate with probability 1 [BO83]. It is known how to convert any deterministic obstruction-free algorithm into a randomized wait-free algorithm against an oblivious adversary [GHHW13].

This recent progress on space complexity of consensus overviewed in previous sections highlights how little is known about the space complexity of k-set agreement. The best obstruction-free algorithms require $n - k + 1$ registers [Zhu15, BRS15], and work even for anonymous processes. Bouzid et al. [BRS15] also give an x-obstruction-free algorithm that uses $n - k + x$ registers, improving on the $min(n + 2x - k, n)$ space complexity of Delporte-Gallet, Fauconnier, Gafni, and Rajsbaum's algorithm [DGFGR13]. Delporte-Gallet, Fauconnier, Kuznetsov, and Ruppert [DGFKR15] proved that it is impossible to solve k-set agreement using 1 register, but nothing better is known. For anonymous processes, they also proved a lower bound of $\sqrt{x(\frac{n}{k}-2)}$ for x-obstruction-free algorithms, which still leaves a polynomial gap between the lower and upper bounds.

In this thesis, we prove a space lower bound of $n-k+1$ registers for solving n-process k-obstruction-free k-set agreement. This also implies an improved lower bound of n registers for consensus. The technique we develop for proving this result stems from simple intuition from combinatorial topology. We build a novel algorithmic reduction to the impossibility of solving wait-free k-set agreement via a simulation, in which the simulated processes run specific types of executions, reminiscent of executions used by the adversary in Zhu's lower bound [Zhu16]. We believe that the simulation can be generalized to lead to a space lower bound of $\lfloor\frac{n-x+1}{k+1-x}\rfloor+1$ registers for solving n-process x-obstruction-free k-set agreement. This generalized result is provided in the full version of our paper [EGZ17].

Intuition: There are good reasons why proving lower bounds on the number of registers for k-set agreement may be substantially more difficult than for consensus. The covering technique due to Burns and Lynch [BL93] used in virtually all space lower bounds (in particular for consensus [FHS98, Gel15, Zhu16]), resembles the FLP impossibility argument in the following sense: starting from a suitable initial configuration, an execution is repeatedly extended to reach subsequent configurations that satisfy a *valency* invariant about the number of values that can still be returned. Additionally, in covering lower bounds, the algorithm is forced to use more and more registers. This approach fails for k-set agreement. On a high level, the impossibility results for k-set agreement consider some representation (a simplicial complex, or a multi-graph) of all possible process states in all possible executions. Then, Sperner's Lemma [Sol49] is used to prove that, roughly speaking, for any given number of steps, there exists an execution leading to a configuration in which processes have not agreed on fewer than $k+1$ values.

Informally, the latter type of argument can be thought as "global", since it proves the existence of a bad execution based on the global representation of all process states in all possible executions. The former type of argument can be thought as "local", since it considers a fixed configuration satisfying certain invariants, some indistinguishable configurations, and inductively extends the execution to reach another configuration.

As an illustrating example, consider the iterated immediate snapshot (IIS) model of [BG97], where processes participate in a sequence of rounds, each associated with a single-writer snapshot object. In round i, each process performs two steps: it updates S_i with its full state and then performs a scan of S_i. Consider r-*synchronized* configurations in which every process has completed round r, but not yet started round $r + 1$. It is known that the global representation of all reachable r-synchronized configurations corresponds to the r-th iterated standard chromatic subdivision of an n-simplex. The vertices represent possible process states after r rounds of IIS and every face of the subdivided simplex corresponds to a reachable r-synchronized configuration. Since we consider r-synchronized configurations, the value that a process returns in a solo execution starting from such a configuration only depends on its state (because it will not see any other process in subsequent rounds). We call this value the *solo valency* of the process in that state and we use it to color the corresponding vertex. Sperner's Lemma is applicable to the original (input) complex, since all initial configurations satisfy the following *boundary condition*: for any set of processes P, any value returned in a P-only execution from that configuration is the solo valency of some process in P. This guarantees that one of the faces in the r-th subdivision will correspond to a "good" configuration, i.e. it will have the solo valency property that we need. However, for faces of the subdivision, the boundary condition might not hold. Thus, this "global" argument cannot be used to extend a "good" r-synchronized configuration to a "good" r'-synchronized configuration by $r' - r$ additional IIS rounds. This is a serious challenge for local arguments.

On the other hand, we do not know enough about the topological representation of algorithms that are x-obstruction-free, or use fewer than n multi-writer registers [HKR13]. There is ongoing work to develop a more general theory [GKM14, SHG16, GHKR16]. However, as of now we do not know enough to prove a space lower bound via a topological argument. Known approaches that do not explicitly use topology [AC11, AP12] also do not overcome these problems.

Technical Overview: We start in a constrained setting and consider algorithms for k-set agreement that satisfy k-obstruction-freedom and, additionally, a certain boundary condition. Confirming our intuition, in this setting we are able to successfully derive an $n-k-1$ space lower bound using a covering argument. Informally, this is a "local" argument, which inductively extends an execution to reach another configuration that has $k+1$ valencies. To apply Sperner's Lemma directly, we would have to develop a characterization of k-obstruction-free solvability using multi-writer registers, which we do not know how to do. Instead, we construct simulations that serve as algorithmic reductions to the impossibility of k-set agreement. If $k+1$ valencies cannot be maintained or if processes return, then we design an algorithm for $k+1$ processes to solve wait-free k-set agreement via a suitable simulation This can be viewed as an indirect, black-box application of Sperner's Lemma.

We then use an insight from the impossibility results for k-set agreement as they apply Sperner's lemma only once, from the initial configuration. The impossibility tells us that there is an infinite execution where $k+1$ valencies are maintained, we just cannot incrementally build it. Instead, we design a simulation for $k+1$ processes to solve k-set agreement based on a given protocol Π for n processes, such that, either the simulation yields a wait-free solution to the task, which is impossible, or the simulating processes perform an arbitrarily large number of simulated steps. We control the scheduling of simulated steps, and by using the scheduling algorithm from the space lower bound adversary of [Zhu16], the simulating processes marshall their simulated processes such that they cover mode and more registers. In order to stay consistent, the processes need to simulate block writes, and we accomplish this by generalizing multi-writer register implementation of [DS97].

1.2.5 Chapter Outline

We will first show the proof of the $\Omega(n)$ anonymous space lower bound.

Then, we describe our new hierarchy from [EGSZ16] based on the space complexity of solving obstruction-free consensus. For various instruction sets $\mathcal{I}$, we provide upper and lower bounds on $\mathcal{SP}(\mathcal{I}, n)$, the minimum number of memory locations (supporting $\mathcal{I}$) needed to solve obstruction-free n-valued consensus among $n \geq 2$ processes (abbreviated as *n-consensus*). The results are summarized in Table 1.1. For a given set of instructions $\mathcal{I}$, $\mathcal{SP}(\mathcal{I}, n)$ is a function of n, that takes values on positive integers. We will present detailed proofs of some of these results here.

Instructions $\mathcal{I}$	$\mathcal{SP}(\mathcal{I}, n)$
$\{$*read*$()$, *test-and-set*$()\}$, $\{$*read*$()$, *write*$(1)\}$	∞
$\{$*read*$()$, *test-and-set*$()$, *reset*$()\}$, $\{$*read*$()$, *write*(1), *write*$(0)\}$	$n-1$ (lower), $O(n \log n)$ (upper)
$\{$*read*$()$, *write*$(x)\}$	$n-1$ (lower), n (upper)
$\{$*read*$()$, *swap*$(x)\}$	$\Omega(\sqrt{n})$ (lower), $n-1$ (upper)
$\{\ell$-*buffer-read*$()$, ℓ-*buffer-write*$(x)\}$	$\lceil \frac{n-1}{\ell} \rceil$ (lower), $\lceil \frac{n}{\ell} \rceil$ (upper)
$\{$*read*$()$, *write*(x), *increment*$()\}$ $\{$*read*$()$, *write*(x), *fetch-and-increment*$()\}$	2 (lower), $O(\log n)$ (upper)
$\{$*read-max*$()$, *write-max*$(x)\}$	2
$\{$*compare-and-swap*$(x, y)\}$ $\{$*read*$()$, *set-bit*$(x)\}$ $\{$*read*$()$, *add*$(x)\}$, $\{$*read*$()$, *multiply*$(x)\}$ $\{$*fetch-and-add*$(x)\}\}$, $\{$*fetch-and-multiply*$(x)\}$	1

Table 1.1: Space Hierarachy

Consider the instructions

- *multiply*(x), which multiplies the number stored in a memory location by x and returns nothing, and
- *add*(x), which adds x to the number stored in a memory location and returns nothing. and
- *set-bit*(x), which sets bit x of a memory location to 1 and returns nothing.

We show that one memory location supporting *read*$()$ and one of these instructions can be used to solve n-consensus. The idea is to show that these instruction sets can implement n counters in a single location and then use a *racing counters* algorithm [AH90], adjusted to fit our needs.

We will also show that a single memory location supporting the set of instructions $\{$*read*$()$, *write*(x), *fetch-and-increment*$()\}$ cannot be used to solve n-consensus, for $n \geq 3$. On the positive side, we also present an algorithm for solving n-consensus using $O(\log n)$ such memory locations.

Next, we will introduce a family of buffered read and buffered write instructions $\mathcal{B}_\ell$, for $\ell \geq 1$, and show how to solve n-consensus using $\lceil \frac{n}{\ell} \rceil$ memory locations supporting these instructions. Extending Zhu's $n-1$ lower bound [Zhu16], we also prove that $\lceil \frac{n-1}{\ell} \rceil$ such memory locations are necessary, which is tight except when $n-1$ is divisible by ℓ. Moreover, we will show a surprising result that the preceding lower bound holds within a factor of two even in the presence of atomic multiple assignment. Multiple assignment can be implemented by simple transactions, so our result implies that such transactions cannot significantly reduce space complexity. The proof further extends the techniques of [Zhu16] via a combinatorial argument, which we hope will be of independent interest.

Then, we will describe our practical lock-free universal construction that uses only atomic *read*, *xor*, *decrement*, and *fetch-and-increment* instructions. We will conclude Chapter 3 by our simulation-based argument for the space complexity of k-set agreement.

1.3 On Message Passing

In the asynchronous shared-memory model, (almost) tight complexity bounds are known for randomized implementations of tasks such as consensus [AC08], mutual exclusion [HW09, HW10, GW12b], renaming [AACH$^+$14], and task allocation [BKRS96, ABGG12].

Less is known about the complexity of randomized distributed tasks in the *asynchronous message-passing* model[4]. In message-passing, a set of n processors communicate via point-to-point channels. Communication is *asynchronous*, i.e., messages can be arbitrarily delayed. Further, the system is controlled by an *adaptive adversary*, which sees the contents of messages and local state, and can choose to crash less than half of the participants at any point during the computation.

We focus on *test-and-set (leader election)*, which is the distributed equivalent of a

[4]Simulations between the two models exist [ABND95], but their complexity overhead is considerable.

tournament: each process must return either a *winner* or a *loser* indication, with the property that *exactly one* process may return *winner.* The time complexity of leader election against a strong adversary is a major open problem. No lower bounds are known. The fastest known solution is more than two decades old [AGTV92], and is a *tournament tree*: pair up the participants into two-processor "matches," decided by two-processor randomized consensus; winners continue to compete, while losers drop out, until a single winner prevails. The time complexity is logarithmic, as the winner has to communicate at each tree level. Despite significant interest and progress on this problem in weaker adversarial models [AAG+10, AA11, GW12a], the question of whether a tournament is optimal as a way to elect a leader against a strong adversary was not known.

In the master's thesis of the author [Gel14], it was shown that this is not the case in message-passing, by describing an algorithm, called PoisonPill, that solves test-and-set in expected $O(\log \log n)$ time. The general structure is rather simple: computation occurs in *phases*, where each phase is designed to drop as many participants as possible, while ensuring that at least one processor survives. The main algorithmic idea was a way to *hide* the processor coin flips during the phase, handicapping the adaptive adversary. In each phase, each processor first takes a "poison pill" (moves to *commit* state), and broadcasts this to all other processors. The processor then flips a biased local coin to decide whether to drop out of contention (*low* priority) or to take an "antidote" (*high* priority), broadcasts its new state, and checks the states of other processors. Crucially, if it has flipped low priority, and sees *any other processor* either in *commit* state or in *high priority* state, the processor returns *lose.* Otherwise, it survives to the next phase.

The above mechanics guarantee at least one survivor (in the unlikely event where all processors flip *low* priority, they all survive), but can lead to few survivors in each phase. The insight is that, to ensure many survivors, the adversary must examine the processors' coin flips. But to do so, the adversary must first allow it to take the poison pill (state *commit*). Crucially, any low-priority processor observing this *commit* state automatically drops out. We prove that, because of this catch-22, the adversarial scheduler can do no more than to let processors execute each phase sequentially, one-by-one, hoping that the first processor flipping high priority, which eliminates all later low-priority participants, comes as late as possible in the sequence. We bias the flips such that a group of at most square root participants survive because they flipped high priority, and square root participants survive because they did not observe any high priority. This choice of bias seems hard to improve, as it yields the perfect balance between the sizes of the two groups of survivors.

In Chapter 4, we will describe our algorithm from [AGV15a], where we further improved the time complexity of the test-and-set algorithm using a second algorithmic idea that breaks the above-mentioned roadblock. Consider two extreme scenarios for a phase: first when all participants communicate with each other, leading to similar views and second, when processors see fragmented views, observing just a subset of other processors. In the first case, each processor can safely set a low probability of surviving. This does not work in the second case since processor views have a lot of variance. We exploit this variance to break symmetry. Our technical argument combines these two strategies such that we obtain at most $O(\log^2 n_r)$ expected survivors in a phase, under *any* scheduling. The resulting algorithm is *adaptive*, meaning that, if $k \leq n$ processors participate, its complexity becomes $O(\log^* k)$.

Chapter 2

Population Protocols

We start the chapter by formally defining the model.

2.1 Model

A *task* in the population protocol model is specified by a finite set of input states I, and a finite set of output symbols, O. The predicate corresponding to the task maps any input configuration onto an allowable set of output symbols. We instantiate this definition for majority and leader election below.

A *population protocol* $\mathcal{P}_k$ with k states is defined by a triple $\mathcal{P}_k = (\Lambda_k, \delta_k, \gamma_k)$. Λ_k is the set of *states* available to the protocol, satisfying $I \subseteq \Lambda_k$ and $|\Lambda_k| = k$. The protocol consists of a set of state transitions of the type

$$A + B \rightarrow C + D,$$

defined by the protocol's state transition function $\delta_k : \Lambda_k \times \Lambda_k \rightarrow \Lambda_k \times \Lambda_k$. Finally, $\gamma_k : \Lambda_k \rightarrow O$ is the protocol's output function.

This definition extends to protocols which work for *variable* number of states: in that case, the population protocol $\mathcal{P}$ will be a sequence of protocols $\mathcal{P}_m, \mathcal{P}_{m+1}, \ldots,$ where m is the minimal number of states which the protocol supports.

In the following, we will assume a set of $n \geq 2$ agents, interacting pairwise. Each of the agents, or nodes, executes a deterministic state machine, with states in the set Λ_k. The *legal initial configurations* of the protocol are exactly configurations where each agent starts in a state from I. Once started, each agent keeps updating its state following interactions with other agents, according to a transition function δ_k. Each *execution step* is one interaction between a pair of agents, selected to interact uniformly at random from the set of all pairs. The agents in states S_1 and S_2 transition to states given by $\delta_k(S_1, S_2)$ after the interaction.

Configurations: Agents are *anonymous*, so any two agents in the same state are identical and interchangeable. Thus, we represent any set of agents simply by the *counts of agents* in every state, which we call a *configuration.* More formally, a *configuration* c is a function $c : \Lambda_k \to \mathbb{N}$, where $c(S)$ represents the *number of agents in state* S *in configuration* c. We let $|c|$ stand for the sum, over all states $S \in \Lambda_k$, of $c(S)$, which is the same as the total number of agents in configuration c. For instance, if c is a configuration of all agents in the system, then c describes the global state of the system, and $|c| = n$.

We say that a configuration c' is *reachable* from a configuration c, denoted $c \Longrightarrow c'$, if there exists a sequence of consecutive steps (interactions from δ_k between pairs of agents) leading from c to c'. If the transition sequence is p, we will also write $c \Longrightarrow_p c'$. We call a configuration c the *sum of configurations* c_1 and c_2 and write $c = c_1 + c_2$, when $c(S) = c_1(S) + c_2(S)$ for all states $S \in \Lambda_k$.

Majority: In the *majority problem*, nodes start in one of two initial states $A, B \in I$. The output set is $O = \{Win_A, Win_B\}$, where, intuitively, an initial state wins if its initial count is larger than the other state's. Formally, given an initial configuration i_n, let $\epsilon n = |i_n(A) - i_n(B)|$ denote the *discrepancy*, i.e. initial relative advantage of the majority state.

We say that a configuration c *correctly outputs the majority decision* for i_n, when for any state $S \in \Lambda_k$ with $c(S) > 0$, if $i_n(A) > i_n(B)$ then $\gamma_k(S) = Win_A$, and if $i_n(B) > i_n(A)$ then $\gamma_k(S) = Win_B$. (The output in case of an initial tie can be arbitrary.) A configuration c has a *stable correct majority decision* for i_n, if for all configurations c' with $c \Longrightarrow c'$, c' correctly outputs the majority decision for i_n.

A population protocol $\mathcal{P}_k$ *stably computes majority decision* from i_n within ℓ steps with probability $1-\phi$, if, with probability $1-\phi$, any configuration c reachable from i_n by the protocol with $\geq \ell$ steps has a stable correct majority decision. In this thesis, we consider the *exact* majority task, as opposed to *approximate* majority [AAE08b], which allows nodes to produce the wrong output with some probability.

Leader Election: In the *leader election* problem, $I = \{A\}$, and in the initial configuration i_n all agents start in the same initial state A. The output set is $O = \{\mathit{Win}, \mathit{Lose}\}$. Intuitively, a single node should output Win, while the others should output Lose.

We say that a configuration c *has a single leader* if there exists some state $S \in \Lambda_n$ with $\gamma_n(S) = \mathit{Win}$ and $c(S) = 1$, such that for any other state $S' \neq S$, $c(S') > 0$ implies $\gamma_n(S') = \mathit{Lose}$. A configuration c of n agents has a *stable leader*, if for all c' reachable from c, it holds that c' has a single leader.

A population protocol $\mathcal{P}_k$ *stably elects a leader* within r steps with probability $1-\phi$, if, with probability $1-\phi$, any configuration c reachable from i_n by the protocol within $\geq r$ steps has a stable leader.

Complexity Measures: The above setup considers sequential interactions; however, interactions between pairs of distinct agents are independent, and are usually considered as occurring in parallel. It is customary to define one unit of *parallel time* as n consecutive steps of the protocol.

A population protocol $\mathcal{P}$ stably elects a leader using $s(n)$ states in time $t(n)$ if, for all sufficiently large n, the expected number of steps for protocol $\mathcal{P}_{s(n)}$ (with $s(n)$ states) to stably elect a leader from the initial configuration, divided by n, is $t(n)$. We call $s(n)$ the *state complexity* and $t(n)$ the *time complexity* (or stabilization time) of the protocol. For the majority problem, the complexity measures might also depend on ϵ. Thus, $\mathcal{P}$ having state complexity $s(n, \epsilon)$ and time complexity $t(n, \epsilon)$ means that for sufficiently large n, $\mathcal{P}_{s(n,\epsilon)}$ stabilizes to the correct majority decision in expected time $t(n, \epsilon)$ for all ϵ. If the expected time is finite, then we say that population protocol stably elects a leader (or stably computes majority decision).

Monotonicity: The above definition of population protocols only requires that for any n, there is just one protocol $\mathcal{P}_{s(n)}$ that stabilizes fast for n agents. In particular, notice that, so far, we did not constrain how protocols $\mathcal{P}_k$ with different number of states k are related to each other.

Additionally, we would like our protocols to be *monotonic*, meaning that a population protocol with a certain number of states that solves a task for n agents should not be slower when running with $n' < n$ agents. Formally, a monotonic population protocol $\mathcal{P}$ stably elects a leader with $s(n)$ states in time $t(n)$, if there exists a sufficiently large constant d, such that for all $n \geq d$, protocol $\mathcal{P}_{s(n)}$ stably elects a leader from the initial configuration $i_{n'}$ of n' agents, for any n' with $d \leq n' \leq n$, in expected parallel time $t(n)$.

A monotonic population protocol $\mathcal{P}$ stably computes majority decision with $s(n, \epsilon)$ states in time $t(n, \epsilon)$, if there exists a sufficiently large constant d, such that for all $n \geq d$, $\mathcal{P}_{s(n,\epsilon)}$ stably computes majority decision from the initial configuration $i_{n'}$ of n' agents with discrepancy $\epsilon' n'$, for any n' with $d \leq n' \leq n$ and $\epsilon' \geq \epsilon$, in expected parallel time $t(n, \epsilon)$.

Weak Monotonicity: We will also consider a different, weaker version of monotonicity that is satisfied when a protocol used for more nodes never has less states. In particular, for leader election, we only require that the state complexity function $s(n)$ be monotonically non-decreasing for all sufficiently large n. For majority, we require that for any fixed ϵ, and sufficiently large n, $s(n, \epsilon)$ be monotonically non-decreasing, and additionally, that $\mathcal{P}_{s(n,\epsilon)}$ should also correctly solve majority for n agents with discrepancy $\epsilon' n$, where $\epsilon' > \epsilon$, but with arbitrary finite stabilization time.

Notice that weak monotonicity is actually far too weak for practical algorithms, as it does not demand a protocol that works for a large number of nodes to also work for smaller number of nodes. However, we will be able to prove certain strong lower bounds even under this extremely non-restrictive condition.

Output Dominance: Our conditional lower bound will make the following additional assumption on the output properties of population protocols for majority:

Definition 2.1.1 (Output Dominance)**.** *For any population protocol $\mathcal{P}_k \in \mathcal{P}$, let c be a configuration with a stable majority decision. Let let c' be another configuration, such that for any state $S \in \Lambda_k$, if $c'(S) > 0$, then $c(S) > 0$. Then, for any configuration c'' such that $c' \Longrightarrow c''$, if c'' has a stable majority decision, then this decision is the same as in c.*

Intuitively, output dominance says that, if we change the *counts* of states in any configuration c with a stable output, then the protocol will still stabilize to the same output decision. In other words, the protocol cannot swap output decisions from a stable configuration if the count of some states changes. To our knowledge, all known techniques for achieving exact majority in population protocols satisfy this condition. In fact, known algorithms satisfy the following stronger condition, which we call *output closedness*:

Definition 2.1.2 (Output Closedness)**.** *Consider any $\mathcal{P}_k \in \mathcal{P}$ and configuration c, such that all nodes in c support the same majority decision. That is, for all $S \in \Lambda_k$ with $c(S) > 0$, $\gamma_k(S) = WIN_X$ for a fixed $X \in \{A, B\}$. Then, for any c' with $c \Longrightarrow c'$, if c' has a stable majority decision, then this decision is WIN_X.*

As mentioned, output closedness implies output dominance.

Lemma 2.1.3. *If a population protocol $\mathcal{P}$ satisfies output closedness, then it satisfies output dominance.*

Proof. For any $\mathcal{P}_k$, consider an arbitrary configuration c that has a stable majority decision WIN_X for $X \in \{A, B\}$. Let c' be another configuration, such that for any $S \in \Lambda_k$, if $c'(S) > 0$ then $c(S) > 0$. Since c has a stable output, for all $S \in \Lambda_k$, if $c'(S) > 0$, then we also have $\gamma_k(S) = WIN_X$. By output closedness, for any c'' such that $c' \implies c''$, if c'' has a stable majority decision, then it is WIN_X. This is the same as the decision in c, completing the proof. $\square$

Next, to exemplify the model, we describe our leader-minion algorithm from [AG15] and prove its stabilization guarantees.

2.2 Leader-Minion Algorithm

To familiarize the reader with populations protocols, in this section, we describe our LM leader election algorithm from [AG15]. It is a simple algorithm that first achieved polylogarithmic state and time complexity for leader election. The algorithm has an integer parameter $m > 0$, which we set to $\Theta(\log^3 n)$. Each state corresponds to an integer value from the set $\{-m, -m+1, \ldots, -2, -1, 1, 2, m-1, m, m+1\}$. Respectively, there are $2m+1$ different states. We will refer to states and values interchangeably. All nodes start in the same state corresponding to value 1.

The algorithm, specified in Figure 2-1, consists of a set of simple deterministic update rules for the node state. In the pseudocode, the node states before an interaction are denoted by x and y, while their new states are given by x' and y'. All nodes start with value 1 and continue to interact according to these simple rules. We prove that all nodes except one will stabilize to negative values, and that stabilization is fast with high probability. This solves the leader election problem since we can define γ as mapping only positive states to Win (a leader). (Alternatively, γ that maps only two states with values m and $m+1$ to WIN would also work, but we will work with positive leader states for the simplicity of presentation.)

Since positive states translate to being a leader according to γ, we call a node a *contender* if it has a positive value, and a *minion* otherwise. We present the algorithm in detail below. The state updates (i.e. the transition function δ) of the LM algorithm are completely symmetric, that is, the new state x' depends on x and y (lines 2-4) exactly as y' depends on y and x (lines 5-7).

If a node is a contender and has absolute value not less than the absolute value

```
Parameters:
$m$, an integer $> 0$, set to $\Theta(\log^3 n)$
State Space:
$LeaderStates = \{1, 2, \ldots, m-1, m, m+1\}$,
$MinionStates = \{-1, -2, \ldots, -m+1, -m\}$,
Input: States of two nodes, $x$ and $y$
Output: Updated states $x'$ and $y'$
Auxiliary Procedures:
$\textit{is-contender}(x) = \begin{cases} \mathsf{true} & \text{if } x \in LeaderStates; \\ \mathsf{false} & \text{otherwise.} \end{cases}$
$\textit{contend-priority}(x, y) = \begin{cases} m & \text{if } \max(|x|, |y|) = m+1; \\ \max(|x|, |y|) + 1 & \text{otherwise.} \end{cases}$
$\textit{minion-priority}(x, y) = \begin{cases} -m & \text{if } \max(|x|, |y|) = m+1; \\ -\max(|x|, |y|) & \text{otherwise.} \end{cases}$
1 procedure update$\langle x, y\rangle$
2     if $\textit{is-contender}(x)$ and $|x| \geq |y|$ then
3         $x' \leftarrow \textit{contend-priority}(x, y)$
4     else $x' \leftarrow \textit{minion-priority}(x, y)$
5     if $\textit{is-contender}(y)$ and $|y| \geq |x|$ then
6         $y' \leftarrow \textit{contend-priority}(x, y)$
7     else $y' \leftarrow \textit{minion-priority}(x, y)$
```

Figure 2-1: The state update rules for the *LM* algorithm.

of the interaction partner, then the node remains a contender and updates its value using the *contend-priority* function (lines 3 and 6). The new value will be one larger than the previous value except when the previous value was $m + 1$, in which case the new value will be m.

If a node had a smaller absolute value than its interaction partner, or was a minion already, then the node will be a minion after the interaction. It will set its value using the *minion-priority* function, to either $-\max(|x|, |y|)$, or $-m$ if the maximum was $m + 1$ (lines 4 and 7).

Values $m + 1$ and m are treated exactly the same way by minions (essentially corresponding to $-m$). These values serve as a binary tie-breaker among the contenders that ever reach the value m, as will become clear from the analysis.

2.2.1 Analysis

Throughout the proof, we call a node *contender* when the value associated with its state is positive, and a *minion* when the value is negative. As previously discussed, we assume that $n > 2$. For presentation purposes, we also consider n to be a power of two.

We first prove that the algorithm never eliminates all contenders and that a configuration with a single contender means that a leader is elected.

Lemma 2.2.1. *There is always at least one contender in the system. Suppose the execution reaches a configuration c with only node v being a contender. Then, v remains a contender (mapped to WIN by γ) in any configuration c' reachable from c, and c' never contains another contender.*

Proof. By the structure of the algorithm, a node starts as a contender and may become a minion during an execution, but a minion may never become a contender. Moreover, an absolute value associated with the state of a minion node can only increase to an absolute value of an interaction partner.

Suppose for contradiction that an execution reaches a configuration $\hat{c}$ where all nodes are minions. Let the maximum absolute value of the nodes be u in $\hat{c}$. Because the minions cannot increase the maximum absolute value in the system, there must have been a contender with value u during the execution before the execution reached $\hat{c}$. For this contender to have become a minion, it must have interacted with another node with an absolute value strictly larger than u. The absolute value of a node never decreases except from $m + 1$ to m, and despite existence of a larger absolute value than u before reaching $\hat{c}$, u was the largest absolute value in $\hat{c}$. Thus, u must be equal to m. But after such an interaction, the second node that was in the state $m + 1$ remains a contender with value m. Before the execution reaching $\hat{c}$, it must also have interacted with yet another node with value $m+1$ in order to become a minion itself. But then, the interaction partner remains a contender with value m and the same reasoning applies to it. Our proof follows by infinite descent.

Consequently, whenever there is a single contender in the system, it must have the largest absolute value. Otherwise, it could interact with a node with a larger absolute value and become a minion, contradicting the above proof that all nodes may never be minions. Due to this invariant, the only contender may never become a minion and we know the minions can never become contenders. □

Now we turn our attention to the stabilization speed (assuming $n > 2$) of the *LM* algorithm. Our goal is bound the number of steps necessary to eliminate all except a single contender. In order for a contender to get eliminated, it must come across a larger value of another contender, the value possibly conducted through a chain of multiple minions via multiple interactions.

We first show by a rumor spreading argument that if the difference between the values of two contenders is large enough, then the contender with the smaller value will become a minion within the next $O(n \log n)$ interactions, with constant probability. Then we use anti-concentration bounds to establish that for any two fixed contenders, given that no absolute value in the system reaches m, after every $O(n \log^2 n)$ interactions the difference between their values is large enough with constant probability.

Lemma 2.2.2. *Consider a configuration c, in which there are two contenders with values u_1 and u_2, where $u_1 - u_2 \geq 4\xi \log n$ for $\xi \geq 8$. Then, after $\xi n \log n$ interactions from c, the node that initially held the value u_2 will be a minion with probability at least $1/24$ (independent of the history of previous interactions leading up to c).*

Proof. We call a node that has an absolute value of at least u_1 an *up-to-date* node, and *out-of-date* otherwise. Initially, at least one node is up-to-date. When there are x up-to-date nodes, the probability that an out-of-date node interacts with an up-to-date node next, increasing the number of up-to-date nodes to $x + 1$, is $\frac{2x(n-x)}{n(n-1)}$. By a Coupon Collector argument, the expected number of steps until every node is up-to-date is $\sum_{x=1}^{n-1} \frac{n(n-1)}{2x(n-x)} \leq \frac{(n-1)}{2} \sum_{x=1}^{n-1} \left(\frac{1}{x} + \frac{1}{n-x}\right) \leq 2n \log n$.

By Markov's inequality, the probability that not all nodes are up-to-date after $\xi n \log n$ interactions is at most $2/\xi$. Hence, expected number of up-to-date nodes after $\xi n \log n$ interactions is at least $\frac{n(\xi-2)}{\xi}$. Let q be the probability that the number of up-to-date nodes after $\xi n \log n$ interactions is at least $\frac{n}{3}+1$. We have $qn+(1-q)(\frac{n}{3}+1) \geq$

$\mathbb{E}[Y] \geq \frac{n(\xi-2)}{\xi}$, which implies $q \geq \frac{1}{4}$ for $n > 2$ and $\xi \geq 8$.

Hence, with probability at least 1/4, at least $n/3 + 1$ are nodes are up to date after $\xi n \log n$ interactions from configuration c. By symmetry, the $n/3$ up-to-date nodes except the original node are uniformly random among the other $n-1$ nodes. Therefore, any given node, in particular the node that had value u_2 in c has probability at least $1/4 \cdot 1/3 = 1/12$ to be up-to-date after $\xi n \log n$ interactions. When the node that was holding value u_2 in c becomes up-to-date and gets an absolute value of at least u_1 from an interaction, it must become a minion by the structure of the algorithm if its value before this interaction was still strictly smaller than u_1. Thus, we only need to show that the probability of selecting the node that initially had value u_2 at least $4\xi \log n$ times (so that its value can reach u_1) during these $\xi n \log n$ interactions is at most 1/24. The claim then follows by Union Bound.

In each interaction, the probability to select this node (that initially held u_2) is $2/n$. Let us describe the number of times it is selected in $\xi n \log n$ interactions by considering a random variable $Z \sim \text{Bin}(\xi n \log n, 2/n)$. By Chernoff Bound, the probability being selected at least $4\xi \log n$ times is at most:

$$\Pr\left[Z \geq 4\xi \log n\right] \leq \exp\left(-\frac{2\xi}{3}\log n\right) \leq \frac{1}{n^{2\xi/3}} \leq \frac{1}{24}$$

finishing the proof. □

Next, we show that, after $\Theta(n \log^2 n)$ interactions, the difference between the values of any two given contenders is high, with a reasonable probability.

Lemma 2.2.3. *For an arbitrary configuration c, fix two conteders in c and a constant $\xi \geq 1$. Let c' be a configuration reached after $32\xi^2 n \log^2 n$ interactions from c.*

If absolute values of all nodes are strictly less than m at all times before reaching c', then, with probability at least $\frac{1}{24} - \frac{1}{n^{8\xi}}$, in c', either at least one of the two fixed nodes have become minions, or their absolute values differ by at least $4\xi \log n$.

Proof. Suppose no absolute value reaches m at any point before reaching c' and that the two fixed nodes are still contenders in c'. We need to prove that the difference of values is large enough.

Consider the $32\xi^2 n \log^2 n$ interactions following c. If an interaction involves exactly one of the two fixed nodes, we call it a *spreading*. For each interaction, probability of it being spreading is $\frac{4(n-2)}{n(n-1)}$, which for $n > 2$ is at least $2/n$. So, we can describe the number of spreading interactions among the $32\xi^2 n \log^2 n$ steps by considering a random variable $X \sim \text{Bin}(32\xi^2 n \log^2 n, 2/n)$. By Chernoff Bound, the probability of having no more than $32\xi^2 \log^2 n$ spreading interactions is at most

$$\Pr\left[X \leq 32\xi^2 \log^2 n\right] \leq \exp\left(-\frac{64\xi^2 \log^2 n}{2^2 \cdot 2}\right) < \frac{1}{n^{8\xi}},$$

Let us from now on focus on the high probability event that there are at least $32\xi^2 \log^2 n$ spreading interactions between c and c', and prove that the desired difference will be large enough with probability $\frac{1}{24}$. This implies the claim by Union Bound with the above event (since for $n > 2$, $\frac{1}{n^{8\xi}} < \frac{1}{24}$ holds).

We assumed that both nodes remain contenders up until c'. Hence, in each spreading interaction, a value of exactly one of them, with probability $1/2$ each, increases by one. Let us call the fixed nodes V_1 and V_2, and suppose the value of V_1 was not less than the value of V_2 in c. Let us now focus on the sum Y of k independent uniformly distributed ± 1 Bernoulli trials x_i with $1 \leq i \leq k$, where each trial corresponds to a spreading interaction and outcome $+1$ means that the value of V_1 increased, while -1 means that the value of V_2 increased. In this terminology, we are done if we show that $\Pr[Y \geq 4\xi \log n] \geq \frac{1}{24}$ for $k \geq 32\xi^2 \log^2 n$ trials.

However, we have that:

$$\Pr[Y \geq 4\xi \log n] \geq \frac{\Pr[|Y| \geq 4\xi \log n]}{2} = \frac{\Pr[|Y^2| \geq 16\xi^2 \log^2 n]}{2} \tag{2.2.1}$$

$$\geq \frac{\Pr[|Y^2| \geq k/2]}{2} = \frac{\Pr[|Y^2| \geq \mathbb{E}[Y^2]/2]}{2} \tag{2.2.2}$$

$$\geq \frac{1}{2^2 \cdot 2} \frac{\mathbb{E}[Y^2]^2}{\mathbb{E}[Y^4]} \geq \frac{1}{24} \tag{2.2.3}$$

where 2.2.1 follows from the symmetry of the sum with regards to the sign, that is, from $\Pr[Y > 4\xi \log n] = \Pr[Y < -4\xi \log n]$. For 2.2.2 we have used that $k \geq$

$32\xi^2 \log^2 n$ and $\mathbb{E}[Y^2] = k$ (more about this below). Finally, to get 2.2.3 we use Paley-Zygmund inequality and the fact that $\mathbb{E}[Y^4] = 3k(k-1)+k \le 3k^2$. Evaluating $\mathbb{E}[Y^2]$ and $\mathbb{E}[Y^4]$ is simple by using the definition of Y and the linearity of expectation. The expectation of each term then is either 0 or 1 and it suffices to count the number of terms with expectation 1, which are exactly the terms where each multiplier is raised to an even power. $\square$

We are ready to prove the stabilization speed with high probability

Theorem 2.2.4. *There exists a constant α, such that for any constant $\beta \ge 3$ following holds: If we set $m = \alpha\beta \log^3 n = \Theta(\log^3 n)$, the algorithm elects a leader (i.e. reaches a configuration with a single contender) in at most $O(n \log^3 n)$ steps, i.e. in parallel time $O(\log^3 n)$, with probability at least $1 - 1/n^\beta$.*

Proof. Let us fix $\xi \ge 8$ large enough, such that for some constant p

$$\frac{1}{24} \cdot \left(\frac{1}{24} - \frac{1}{n^{8\xi}} \right) \ge p. \tag{2.2.4}$$

Consider constants $\beta \ge 3$ and $\alpha = \frac{16}{p} \cdot (33\xi^2)$. We set $m = \alpha\beta \log^3 n$ and focus on the first $\frac{\alpha\beta n \log^3 n}{4}$ steps of the algorithm execution. For any fixed node, the probability that it interacts in each step is $2/n$. Let us describe the number of times a given node interacts within the first $\frac{\alpha\beta n \log^3 n}{4}$ steps by considering a random variable $\text{Bin}(\frac{\alpha\beta n \log^3 n}{4}, 2/n)$. By Chernoff Bound, the probability being selected at least $m = \alpha\beta \log^3 n$ times is at most $\exp\left(-\frac{\alpha\beta}{6} \log^3 n\right) \le \frac{1}{n^{\alpha\beta/6}}$. By Union Bound over all n nodes, with probability at least $1 - \frac{n}{n^{\alpha\beta/6}}$, all nodes interact strictly less than m times during the first $\frac{\alpha\beta n \log^3 n}{4}$ interactions.

Let us from now on focus on the above high probability event, which means that all absolute values are strictly less than m during the first $\frac{\alpha\beta n \log^3 n}{4} = \frac{4\beta}{p}(33\xi^2) n \log^3 n$ interactions. For a fixed pair of nodes, we apply Lemma 2.2.3 followed by Lemma 2.4.2 (with parameter ξ) $\frac{4\beta(33\xi^2) n \log^3 n}{p(32\xi^2 n \log^2 n + \xi n \log n)} \ge \frac{4\beta \log n}{p}$ times. Each time, by Lemma 2.2.3, after $32\xi^2 n \log^2 n$ interactions with probability at least $\frac{1}{24} - \frac{1}{n^{8\xi}}$ the nodes end up with values at least $4\xi \log n$ apart. In this case, after the next $\xi n \log n$ interactions, by Lemma 2.4.2, one of the nodes becomes a minion with probability at least 1/24. Since Lemma 2.4.2 is independent from the interactions that precede it, by (2.2.4), each of the $\frac{4\beta \log n}{p}$ times if both nodes were contenders, with probability at least p one of the nodes becomes a minion. The probability that both nodes in a given pair are still contenders after the first $\frac{\alpha\beta n \log^3 n}{4}$ steps is thus at most $(1-p)^{\frac{4\beta \log n}{p}} \leq 2^{-4\beta \log n} < \frac{1}{n^{2\beta}}$. By Union Bound over all $\frac{n(n-1)}{2} < n^2$ pairs, with probability at least $1 - \frac{n^2}{n^{2\beta}}$, for every pair of nodes, one of them is a minion after $\frac{\alpha\beta n \log^3 n}{4}$ interactions. Hence, with this probability, there will be only one contender.

Combining with the conditioned event that none of the nodes interact m or more times gives that after the first $\frac{\alpha\beta n \log^3 n}{4} = O(n \log^3 n)$ interactions there must be a single contender with probability at least $1 - \frac{n^2}{n^{2\beta}} - \frac{n}{n^{\alpha\beta/6}} \geq 1 - \frac{1}{n^\beta}$ for $\beta \geq 3$. A single contender means that leader is elected by Lemma 2.2.1. □

Finally, we prove the expected stabilization bound

Theorem 2.2.5. *There is a setting of parameter m of the algorithm such that $m = \Theta(\log^3 n)$, such that the algorithm elects the leader in expected $O(n \log^3 n)$ steps, i.e. in parallel time $O(\log^3 n)$.*

Proof. Let us prove that from any configuration, the algorithm elects a leader in expected $O(n \log^3 n)$ steps. By Lemma 2.2.1, there is always a contender in the system and if there is only a single contender, then a leader is already elected. Now in a configuration with at least two contenders consider any two of them. If their values differ, then with probability at least $1/n^2$ these two contenders will interact next and the one with the lower value will become a minion (after which it may never be a contender again). If the values are the same, then with probability at least $1/n$, one of these nodes will interact with one of the other nodes, leading to a configuration where the values of our two nodes differ[1], from where in the next step, independently, with probability at least $1/n^2$ these nodes will interact and one of them will become a minion. Hence, unless a leader is already elected, in every two steps, with probability

[1]This is always true, even when the new value is not larger, for instance when the values were equal to $m + 1$, the new value of one of the nodes will be $m \neq m + 1$.

at least $1/n^3$ the number of contenders decreases by 1.

Thus, the expected number of interactions until the number of contenders decreases by 1 is at most $2n^3$. In any configuration there can be at most n contenders, thus the expected number of interactions until reaching a configuration with only a single contender is at most $2(n-1)n^3 \leq 2n^4$ from any configuration.

By Theorem 2.2.4 with $\beta = 4$ we get that with probability at least $1 - 1/n^4$ the algorithm stabilizes after $O(n \log^3 n)$ interactions. Otherwise, with probability at most $1/n^4$ it ends up in some configuration from where it takes at most $2n^4$ expected interactions to elect a leader. The total expected number of steps is therefore also $O(n \log^3 n) + O(1) = O(n \log^3 n)$, i.e. parallel time $O(\log^3 n)$. □

Before we design algorithms that use $O(\log n)$ states we have to develop the required tools. We start by the leaderless phase clock.

2.3 Leaderless Phase Clock

Intuitively, the phase clock works as follows. Each node keeps a local counter, intialized at 0. On each interaction, the two nodes compare their values, and the one with the *lower* counter value increments its local counter. We can use the fact that interactions are uniformly random to obtain that the nodes' counter values are concentrated within an additive $O(\log n)$ factor with respect to the mean, with high probability.

The above procedure has the obvious drawback that, as the counters continue to increment, nodes will need unbounded space to store the values. We overcome this as follows. We fix a period $\Psi = \Theta(\log n)$, and a range value $\rho = \Theta(\log n)$, with $\Psi \gg \rho$. The goal of the algorithm is to maintain a "phase clock" with values between 0 and $\Psi - 1$, with the property that clock at different nodes are guaranteed to be within some interval of range ρ around the mean clock value, with high probability.

We let each phase clock state be V_i, where i is from 0 to $\Psi - 1$ and represents the counter value of the node in state V_i. The update rule upon each interaction is as follows. If *both* nodes have counter values either in $[0, \Psi - \rho - 1]$ or $[\Psi - \rho, \Psi - 1]$, then the node that has the *lower* counter value will increment its local counter. Formally, for any $i \leq j$, with $i, j \in [0, \Psi - \rho - 1]$ or $i, j \in [\Psi - \rho, \Psi - 1]$, we have that

$$V_i + V_j \to V_{i+1} + V_j. \tag{2.3.1}$$

In the second case, one of the node values, say i, is in $[0, \Psi - \rho - 1]$, while the other value, j, is in $[\Psi - \rho, \Psi - 1]$. In this case, we simply increment the level of the node with the *higher* counter value. Formally, when $i \in [0, \Psi - \rho - 1]$ and $j \in [\Psi - \rho, \Psi - 2]$, we have that

$$V_i + V_j \to V_i + V_{j+1}. \tag{2.3.2}$$

Finally, if a node would reach counter value Ψ as the result of the increment, it simply resets to value V_0:

$$V_{\Psi-1} + V_{\Psi-1} \to V_{\Psi-1} + V_0 \text{ and } V_i + V_{\Psi-1} \to V_i + V_0, \ \forall i \in [0, \Psi - \rho - 1]. \tag{2.3.3}$$

2.3.1 Analysis

We will show that counter values stay concentrated within around the mean, so that the difference between the largest and the smallest value will be less than $\rho = O(\log n)$, with high probability. The updates in 2.3.2—2.3.3 allow the algorithm to reset the counter value to 0 periodically, once the values reach a range where inconsistent wrap-arounds become extremely unlikely.

For any configuration c, let $w_\ell(c)$ be the weight of node ℓ, defined as follows. Assume node ℓ is in state V_i. For $i \in [0, \rho]$, if in c there exists some node in state V_j with $j \in [\Psi - \rho, \Psi - 1]$ (i.e. if $\sum_{j \in [\Psi-\rho, \Psi-1]} c(V_j) > 0$), then we have $w_\ell(c) = i + \Psi$. Otherwise, we have $w_\ell(c) = i$. Given this definition, let $\mu(c) = \frac{\sum_{\ell=1}^{n} w_\ell(c)}{n}$ be the mean weight, and $x_\ell(c) = w_\ell(c) - \mu(c)$. Let us also define $G(c)$, the *gap* in configuration c, as $\max_\ell w_\ell(c) - \min_\ell w_\ell(c)$.

From an initial configuration with a gap sufficiently smaller than ρ, we consider the number of steps to reach a configuration with a gap of at least ρ. Our goal is to show that a large number of steps is required with high probability. Our definitions are chosen to ensure the following invariant as long as the gap is not $\geq \rho$ in the execution: The evolution of the values $x_\ell(c)$ is identical to that of an algorithm where there is no wrap-around once the value would reach Ψ.

Let us simplify the exposition by considering the process, where values continue to increase unboundedly. Critically, we notice that this process is now identical to the classical two-choice load-balancing process: consider a set of n bins, whose ball counts are initially 0. At each step t, we pick two bins uniformly at random, and insert a ball into the *less loaded* of the two. Here, let us use $x_\ell(t)$ to represents the number of balls in ℓ-th bin, minus the average number of balls per bin after t steps. For a fixed constant $\alpha < 1$, define the potential function

$$\Gamma(t) = \sum_{\ell=1}^{n} 2\cosh(\alpha x_\ell(t)) = \sum_{\ell=1}^{n} \left(\exp(\alpha x_\ell(t)) + \exp(-\alpha x_\ell(t))\right).$$

Peres, Talwar, and Wieder prove in [PTW15] that:

Lemma 2.3.1 (Theorem 2.9 in [PTW15]). *Given the above process, for any $t \geq 0$,*

$$\mathsf{E}[\Gamma(t+1)|\Gamma(t)] \leq \left(1 - \frac{\alpha}{n}\right)\Gamma(t) + \theta, \tag{2.3.4}$$

where $\alpha < 1$ is a constant from the definition of Γ and $\theta \gg 1$ is a fixed constant.

From here, we can prove the following property of the leaderless phase clock.

Corollary 2.3.2. *Given the above process, the following holds: Suppose c is a configuration with $G(c) \leq \gamma \log n$, for some constant γ. Then, for any constant parameter β, there exists a constant $\gamma'(\beta)$, such that with probability $1 - m/n^\beta$, for each configuration c' reached by the m interactions following c, it holds that $G(C') < \gamma'(\beta)\log n$.*

Proof. We let $\gamma'(\beta) = 2\gamma + \frac{4+2\beta}{\alpha}$, where α is the constant from Lemma 2.3.1, and let $\rho = \gamma'(\beta)\log n$. As discussed in Section 2.3, since we are counting the number of steps from configuration c, where the gap is less than ρ, until the gap becomes $\geq \rho$, we can instead analyze the unbounded two-choice process. In the two choice process,

$\Gamma(0)$ corresponds to the potential in configuration c. By simple bounding, we must have that $\Gamma(0) \leq 2n^{\alpha\gamma+1}$. Assume without loss of generality that $\Gamma(0) = 2n^{\alpha\gamma+1}$.

It has already been established by Lemma 2.3.1 that

$$\mathsf{E}[\Gamma(t+1)|\Gamma(t)] \leq \left(1 - \frac{\alpha}{n}\right)\Gamma(t) + \theta.$$

This implies that $\Gamma(t)$ will always tend to *decrease* until it reaches the threshold $\Theta(n)$[2]., so that its expectation will always be below its level at step 0 (in configuration c).

Hence, we have that, for any $t \geq 0$,

$$\mathsf{E}[\Gamma(t)] \leq 2n^{\alpha\gamma+1}.$$

By Markov's inequality, we will obtain that

$$\Pr[\Gamma(t) \geq n^{\alpha\gamma+2+\beta}] \leq 1/n^{\beta}.$$

It follows by convexity of the exponential and the definition of Γ that for each c',

$$\Pr[G(c') \geq 2(\gamma + (2+\beta)/\alpha)\log n] \leq 1/n^{\beta}.$$

Setting $\rho = \gamma'(\beta) = 2\gamma + \frac{4+2\beta}{\alpha}$ and taking union bound over the above event for m steps following configuration c completes the proof. $\square$

2.4 Phased Majority Algorithm

At a high level, the state space of the algorithm algorithm is partitioned into into *worker*, *clock*, *backup* and *terminator* states. Every state falls into one of these categories, allowing us to uniquely categorize the nodes based on the state they are in. The purpose of *worker* nodes is to reach a consensus on the output decision. The

[2]By applying expectation and telescoping, as in the proof of Theorem 2.10 in [PTW15].

purpose of *clock* nodes is to synchronize worker nodes, enabling a logarithmic state space. The job of *backup* nodes is to ensure correctness via a slower protocol, which is only used with low probability. The *terminator* nodes are there to spread a final majority decision. Every node starts as worker, but depending on state transitions, may become a clock, a backup or a terminator.

The algorithm alternates *cancellation* phases, during which workers with different opinions cancel each other out, and *doubling* phases, during which workers which still have a "strong" opinion attempt to spread it to other nodes. Clock nodes will keep these phases in sync.

State Space: The state of a *worker* node consists of a triple of: (1) an *phase number* in $\{1, 2, \ldots, 2\log n + 1\}$; (2) a *value* $\in \{1, 1/2, 0\}$; (3) its *current preference* WIN_A or WIN_B. The state of a *clock* node consists of a pair (1) *position*, a number, describing the current value of its phase clock, initially 0, and (2) its *current preference* for WIN_A or WIN_B. Backup nodes implement a set of four possible states, which serve as a way to implement the four-state protocol of [DV12, MNRS14]. We use this as a slow but dependable backup in the case of a low-probability error event. There are two terminator states, D_A and D_B. Additionally, every state encodes two bits: the node's original input state (A or B) and a clock-creation boolean flag.

Nodes with input A start in a worker state, with phase number 1, value 1, and preference WIN_A. Nodes with input B start in a similar initial state, but with preference WIN_B. The clock-creation flag is *true* for all nodes, and it means that all nodes could still become clocks. The output of a clock or a worker state is its preference. The output of an backup state is the output of the corresponding state of the 4-state protocol. The output mapping for terminator states is the obvious $\gamma(D_A) = WIN_A$ and $\gamma(D_B) = WIN_B$.

A worker node is *strong* if its current *value* is 1/2 or 1. A worker node with value 0 is *weak*. We say that a worker is *in phase* ϕ if its phase number is ϕ. For the phase clock, we will set the precise value of the parameter $\rho = \Theta(\log n)$ in the next section, during the analysis. The size of the clock will be $\Psi = 4\rho$. Clock states with position in $[\rho, 2\rho)$ and $[3\rho, 4\rho)$ will be labelled as buffer states. We will label states $[0, \rho)$ as *ODD* states, and $[2\rho, 3\rho)$ as *EVEN* states.

We now describe the different interaction types, based on the type of states of the interacting nodes. Pseudocode is given in Figure 2-2 and Figure 2-3.

Backup and Terminator Interactions: When both nodes are backups, they behave as in the 4-state protocol of [DV12, MNRS14]. Backup nodes do not change their type, but cause non-backup interaction partners to change their type to a backup. When a node changes to a backup state, it uses an input state of the 4-state protocol corresponding to its original input.

After an interaction between a terminator node in state D_X with $X \in \{A, B\}$ and a clock or worker node with preference WIN_X, both nodes end up in D_X. However, both nodes end up in backup states after an interaction between D_A and D_B, or a terminator node and a worker/clock node of the opposite preference.

Clock State Update: When two clock nodes interact, they update positions according to the phase clock algorithm described in Section 2.3. They might both change to backup states (a low probability event), if their positions had a gap larger than the maximum allowed threshold ρ of the phase clock. A clock node that meets a worker node remains in a clock state with the same position, but adopts the preference of the interaction partner if the interaction partner was strong.

Worker State Update: Let us consider an interaction between two workers in the same phase. When one worker is weak and the other worker is strong, the preference of the node that was weak always gets updated to the preference of the strong node.

Similar to [AAE08a], there are two types of phases. Odd phases are *cancellation phases*, and even phases are *doubling phases.* In a cancellation phase, if both interacting workers have value 1 but different preferences, then both values are updated to 0, preferences are kept, but if clock-creation flag is *true* at both nodes, then one of the nodes (say, with preference WIN_A) becomes a clock. Its position is set to 0 and its preference is carried over from the previous worker state. This is how clocks are created. In a doubling phase, if one worker has value 1 and another has value 0, then both values are updated to 1/2.

Worker Phase and State Updates: Suppose a worker in phase ϕ meets a clock. The clock does not change its state. If ϕ is odd and the label of the clock's state is $EVEN$, or if ϕ is even and the label is ODD, then the worker enters phase $\phi + 1$. Otherwise, the worker does not change its state.

Suppose two workers meet. If their phase numbers are equal, they interact according to the rules described earlier. When one is in phase ϕ and another is in phase $\phi + 1$, the worker in phase ϕ enters phase $\phi + 1$ (the second worker remains unchanged). When phase numbers differ by > 1, both nodes become backups.

Here is what happens when a worker enters phase $\phi + 1$. When $\phi + 1$ is odd and the node already had value 1, then it becomes a a terminator in state D_X given its preference was WIN_X for $X \in \{A, B\}$. Similarly, if the worker was already in maximum round $\phi = 2 \log n + 1$, it becomes a terminator with its preference. Otherwise, the node remains a worker and sets phase number to $\phi + 1$. If $\phi + 1$ is odd and the node had value 1/2, it updates the value to 1, otherwise, the it keeps the value unchanged.

Clock Creation Flag: As described above, during a cancellation, $\mathsf{clock\text{-}creation}$ flag determines whether one of the nodes becomes a clock instead of becoming a weak worker. Initially, $\mathsf{clock\text{-}creation}$ is set to *true* at every node. We will set a threshold $T_c < \rho$, such that when any clock with $\mathsf{clock\text{-}creation}=\mathit{true}$ reaches position T_c, it sets $\mathsf{clock\text{-}creation}$ to *false*. During any interaction between two nodes, one of which has $\mathsf{clock\text{-}creation}=\mathit{false}$, both nodes set $\mathsf{clock\text{-}creation}$ to *false*. A node can never change $\mathsf{clock\text{-}creation}$ from *false* back to *true*.

2.4.1 Analysis

We take a sufficiently large[3] constant β, apply Corollary 2.3.2 with $\gamma = 29(\beta + 1)$, and take the corresponding $\rho = \gamma'(\beta) \log n > \gamma \log n$ to be the whp upper bound on the gap that occurs in our phase clock (an interaction between two clocks with gap $\geq \rho$ leads to an error and both nodes become backups). We set the clock-creation threshold to $T_c = 23(\beta + 1) \log n < \rho$.

We start by proving some useful properties of the algorithm.

Lemma 2.4.1. *In any reachable configuration of the phased majority algorithm from valid initial configurations, the number of clock nodes is at most $n/2$.*

Proof. n workers start in input states and at most one clock is created per two nodes in these initial worker states. This happens only when two workers in the input states with opposite preferences interact while clock-creation is *true*. However, the values get cancelled, and due to the transition rules, the node that did not become a clock may never re-enter the initial state. Therefore, per each clock created there is one node that will never become a clock, proving the claim. □

Lemma 2.4.2 (Rumor Spreading)**.** *Suppose that in some configuration c, one node knows a rumor. The rumor is spread by interactions through a set of nodes S with $|S| \geq n/2$. Then, the expected number of interactions from c for all nodes in S to know the rumor is $O(n \log n)$. Moreover, for sufficiently large constant β, after $\beta n \log n$ interactions, all nodes know the rumor with probability $1 - n^{-9}$.*

Proof Adopted. This problem, also known as epidemy spreading, is folklore. Analysis follows via coupon collector arguments. The expectation bound is trivial and proved for instance in [AG15], Lemma 4.2.

A formal proof of the high probability claim using techniques from [KMPS95] can for instance be found in [AAE08a]. The fact that rumor spreads through at least half of the nodes affects the bounds by at most a constant factor. To see this, observe that each interaction has a constant probability of being between nodes in $S \cup \{u\}$,

[3]For the purposes of Lemma 2.4.2, described later.

where u is the source of the rumor. Thus, with high probability by Chernoff, constant fraction of interactions actually occur between these nodes and these intaractions act as a rumor spreading on $S \cup \{u\}$. $\square$

Lemma 2.4.3 (Backup). *Let c be a configuration of all nodes, containing a backup node. Then, within $O(n^2 \log n)$ expected intaractions from c, the system will stabilize to the correct majority decision.*

Proof. By Lemma 2.4.2, within $O(n \log n)$ expected interactions all nodes will be in a backup state. That configuration will correspond to a reachable configuration of the 4-state protocol of [DV12, MNRS14], and all remaining interactions will follow this backup protocol. As the nodes have the same input in 4-state protocol as in the original protocol, it can only stabilize to the correct majority decision. Moreover, the 4-state protocol stabilizes in $n^2 \log n$ expected interactions from any reachable configuration, completing the proof. $\square$

We call an execution *backup-free* if no node is ever in a backup state. Next, we define an invariant and use it to show that the system may never stabilize to the wrong majority decision.

Invariant 2.4.4 (Sum Invariant). *For any configuration c define potential function $Q(c)$ as follows. For each worker in c in phase ϕ with value v, if its preference is WIN_A, we add $v \cdot 2^{\log n - \lfloor (\phi-1)/2 \rfloor}$ to $Q(c)$. If its preference is WIN_B, we subtract $v \cdot 2^{\log n - \lfloor (\phi-1)/2 \rfloor}$ from $Q(c)$. Suppose c is reachable from an initial configuration where input $X \in \{A, B\}$ has the majority with advantage ϵn, by a backup-free execution during which no node is ever in a terminator state D_X. If $X = A$, we have $Q(c) \geq \epsilon n^2$, and if $X = B$, then $Q(c) \leq \epsilon n^2$.*

Lemma 2.4.5 (Correctness). *If the system stabilizes to majority decision WIN_X for $X \in \{A, B\}$, then state X had the majority in the initial configuration.*

Proof. Without loss of generality, assume that state A had the majority in the initial configuration (WIN_A is the correct decision). For contradiction, suppose the system stabilizes to the decision WIN_B. Then, the stable configuration may not contain terminators in state D_A or strong workers with preference WIN_A. We show that such configurations are unreachable in backup-free executions.

If any node is in state D_A during the execution, it will remain in D_A unless an error occurs (and nodes change to backup states). In neither of these cases can the system stabilize to decision WIN_B. This is because $\gamma(D_A) = WIN_A$ and in executions where some node enters a backup state, we stabilize to the correct decision by Lemma 2.4.3.

By Invariant 2.4.4, for any configuration C reached by a backup-free execution during which, additionally, no node is ever is state D_A, we have $Q(C) \geq n$. But any configuration C with strictly positive $Q(C)$ contains at least one strong node with preference WIN_A, as desired. □

Hence, in the following, when we show that the system stabilizes, it implies that the decision is correct.

Lemma 2.4.6 (Terminator). *Let c be a configuration of all nodes, containing a terminator node. In backup-free executions, the system stabilizes to the correct majority decision within $O(n \log n)$ interactions in expectation and with high probability. Otherwise, the system stabilizes within $O(n^2 \log n)$ expected intaractions.*

Proof. If there is a backup node in c, then the claim follows from Lemma 2.4.3.

Otherwise, the terminator spreads the rumor, such that the nodes that the rumor has reached are always either in the same terminator state, or in an backup state. By Lemma 2.4.2, this takes $O(n \log n)$ interactions both in expectation and with high probability. If all nodes are in the same terminator state, then the system has stabilized to the correct majority decision by Lemma 2.4.5. Otherwise, there is a backup node in the system, and by Lemma 2.4.3, the system will stabilize within further $O(n^2 \log n)$ expected interactions. □

We derive a lemma about each type of phase.

Lemma 2.4.7 (Cancellation). *Suppose in configuration c every node is either a clock or a worker in the same cancellation phase ϕ (ϕ is odd). Consider executing $8(\beta + 1)n \log n$ interactions from c conditioned on an event that during this interaction* *sequence, no clock is ever in a state with label $EVEN$, and that the phase clock gap is never larger than ρ. Let c' be the resulting configuration. Then, with probability $1 - n^{-\beta}$, in c' it holds that: (1) all strong nodes have the same preference, or there are at most $n/10$ strong nodes with each preference; (2) every node is still a clock, or a worker in phase ϕ.*

Proof. By our assumption, no clock is ever in a state with label *EVEN* during the interaction sequence. This implies that no worker may enter phase $\phi + 1$ or become a terminator. We assumed that the phase clock gap never violates the threshold ρ, and we know all workers are in the same phase, so backups also do not occur.

In configuration c, all workers are in phase ϕ, which is a cancellation phase, and must have values in $\{0, 1\}$. This is true for phase 1, and when a node becomes active in a later cancellation phase, it updates value $1/2$ to 1, so having value $1/2$ is impossible. Thus, the only strong nodes in the system have value 1. As no weak worker or a clock may become strong during these $8(\beta + 1)n \log n$ interactions, the count of strong nodes never increases. The only way the count of strong nodes decreases is when two agents with value 1 and opposite preferences interact. In this case, the count always decreases by 2 (both values become 0 or if clock-creation$=$*true*, one node becomes a clock).

Our claim about the counts then is equivalent to Lemma 5 in [AAE08a] invoked with a different constant (5 instead of 4, as $8(\beta + 1)n \log n > 5(\beta + 1)n \ln n$) and by treating strong nodes with different preferences as $(1, 0)$ and $(0, 1)$. □

Lemma 2.4.8 (Duplication)**.** *Suppose in configuration c every node is either a clock or a worker in the same duplication phase ϕ (ϕ is even). Consider executing $8(\beta + 1)n \log n$ interactions from c conditioned on events that during this interaction sequence (1) no clock is ever in a state with label ODD, (2) the phase clock gap is never larger than ρ, and (3) the number of weak workers is always $\geq n/10$. Let c' be the resulting configuration. Then, with probability $1 - n^{-\beta}$, in c' it holds that: (1) all strong workers have value $1/2$; (2) every node is still a clock, or a worker in phase ϕ.*

Proof. By our assumption, no clock is ever in a state with label *ODD* during the interaction sequence. This implies that no worker may enter phase $\phi + 1$ or become a terminator. We assumed that the phase clock gap never violates the threshold ρ, and we know all workers are in the same phase, so backups also do not occur.

In a duplication phase, workers may not update a state such that their value becomes 1. Consider a fixed strong worker state in configuration c with value 1. By the assumption, probability of an interaction between our fixed node and a weak worker is at least $\frac{n/10}{n(n-1)/2} \geq 1/5n$. If such an interaction occurs, our node's value becomes 1/2. The probability that this does not happen is at most $(1 - 1/5n)^{8(\beta+1)n \log n} \leq (1 - 1/5n)^{5n \cdot (\beta+1) \ln n} = n^{-\beta-1}$. By union bound over at most n nodes, we get that with probability $1 - n^{-\beta}$, no worker will have value 1, as desired. $\square$

Next, we develop a few more tools before proving stabilization guarantees.

Lemma 2.4.9. *Suppose we execute $\alpha(\beta + 1)n \log n$ successive interactions for $\alpha \geq 3/2$. With probability $1 - n^{-\beta}$, no node interacts more than $2\alpha(1 + \sqrt{\frac{3}{2\alpha}})(\beta + 1) \log n$ times in these interactions.*

Proof. Consider a fixed node in the system. In any interaction, it has a probability $2/n$ of being chosen. Thus, we consider a random variable $\text{Bin}(\alpha(\beta + 1)n \log n, 2/n)$, i.e. the number of successes in independent Benoulli trials with probability $2/n$. By Chernoff bound, setting $\sigma = \sqrt{\frac{3}{2\alpha}} \leq 1$, the probability interacting more than $2\alpha(1 + \sigma)(\beta + 1) \log n$ times is at most $1/n^{\beta+1}$. Union bound over n nodes completes the proof.

Notice that the number of interactions trivially upper bounds the number of times a node can go through any type of state transition during these interactions. In particular, the probability that any clock in the system increases its position more than $2\alpha(1 + \sqrt{\frac{3}{2\alpha}})(\beta + 1) \log n$ times during these interactions is $n^{-\beta}$. $\square$

Lemma 2.4.10. *Consider a configuration in which there are between $2n/5$ and $n/2$ clocks, each with a position in $[0, 2\rho)$, and all remaining nodes are workers in the same phase ϕ, where ϕ is odd. Then, the number of interactions before some clock reaches position 2ρ is $O(n \log n)$ with probability $1 - n^{-\beta}$.*

Proof. In this case, until some clock reaches position 2ρ, no backup or terminator nodes may appear in the system. Every interaction between two clocks increases one of them. Therefore, the number of interactions until some clock reaches position 2ρ is upper bounded by the number of interactions until $2\rho n$ interactions are performed between clocks. At each interaction, two clocks are chosen with probability at least $1/9$ (for all sufficiently large n). We are interested in the number of Bernoulli trials with success probability $1/9$, necessary to get $2\rho n$ successes with probability at least $1 - n^{-\beta}$. As we have $\rho = \Theta(\log n)$, this is $O(n \log n)$ by Chernoff bound. □

Lemma 2.4.11. *Let $\delta(c)$ for a configuration c be the number of weak workers minus the number of workers with value* 1*. Suppose that throughout a sequence of interactions from configuration c to configuration c' it holds that (1) all nodes are clocks and workers; and (2) no worker enters an odd phase. Then, $\delta(c') \geq \delta(c)$.*

Proof. We will prove that δ is monotonically non-decreasing for configurations along the interaction sequence from c to c'. Under our assumptions, interactions that affect δ are cancellations and duplications. A cancellation decreases the count of workers with value 1 and increases the count of weak workers, increasing δ of the configuration. A duplication decrements both, the number of workers with value 1, and the number of weak workers, leaving δ unchanged. □

The final theorem below is proved in two parts, in Lemma 2.4.13 and Lemma 2.4.14 that follow.

Theorem 2.4.12. *If the initial majority state has an advantage of ϵn nodes over the minority state, our phased majority algorithm stabilizes to the correct majority decision in $O(\log 1/\epsilon \cdot \log n)$ parallel time, both w.h.p. and in expectation.*

Lemma 2.4.13. *If the initial majority state has an advantage of ϵn nodes over the minority state, our algorithm stabilizes to the correct majority decision in $O(\log 1/\epsilon \cdot \log n)$ parallel time, with high probability.*

Proof. In this argument, we repeatedly consider high probability events, and suppose they occur. In the end, an union bound over all these events gives the desired result.

Consider the first $8(\beta + 1)n \log n$ interactions of the protocol. Initially there are no clocks, and each clock starts with a position 0 and increases its position at most by one per interaction. By Lemma 2.4.9, with probability $1 - n^{-\beta}$, during these interactions no clock may reach position $T_c = 23(\beta + 1) \log n$, as that would require a node to interact more than T_c times. The states of the clock with label $EVEN$ all have position $2\rho \geq 58(\beta + 1) \log n$. Therefore, we can apply Lemma 2.4.7 and get that in the resulting configuration c, with probability $1 - n^{-\beta}$, either all strong workers have the same preference, or the number of strong workers with each preference is at most $n/10$. We will deal with the case when all strong nodes have the same preference later. For now, suppose the number of strong workers with each preference is at most $n/10$. As every cancellation up to this point creates one weak worker and one clock, the number of clocks and weak workers is equal and between $2n/5$ and $n/2$. Thus, for δ defined as in Lemma 2.4.11 we have $\delta(c) \geq n/5 > n/10$. We also know that in configuration c, each node is either a clock that has not yet reached position $T_c = 23(\beta + 1) \log n$ (and thus, also not reached a position with a label $EVEN$), or it is a worker still in phase 1.

By Lemma 2.4.10, with probability at least $1 - n^{-\beta}$, within $O(n \log n)$ interactions we reach a configuration c' where some clock is at a position 2ρ, which has a label $EVEN$. But before this, some clock must first reach position T_c. Consider the first configuration c_1 when this happens. The clock at position T_c would set clock-creation $\leftarrow$ *false*. Notice that from c_1, clock-creation=*false* propagates via rumor spreading, and after the rumor reaches all nodes, no node will ever have clock-creation=*true* again, and no more clocks will be created. By Lemma 2.4.2, this will be the case with high probability[4] in a configuration c_2 reached after $(3/2)\beta n \log n$ interactions from c_1. Moreover, by Lemma 2.4.9, no clock will have reached a position larger than $T_c + 6(\beta + 1) \log n \leq 29(\beta + 1) \log n$ in c_2, which is precisely the quantity $\gamma \log n$ we used as the maximum starting gap when applying Corollary 2.3.2 to determine the ρ of our phase clock. In c_2, all clocks have positions in $[0, 29(\beta + 1) \log n)$, and no more clocks will ever be created. By Lemma 2.4.1 and since the number of

[4]Recall that β was chosen precisely to be sufficiently large for the whp claim of Lemma 2.4.2.

clocks was $\geq 2n/5$ in configuration c, the number of clock nodes is from now on fixed between $2n/5$ and $n/2$ (unless some node becomes a backup or a terminator). Also, the definition of ρ lets us focus on the high probability event in Corollary 2.3.2, that the phase clock gap remains less than ρ during $\Theta(n \log n)$ interactions following c_2.

Since $29(\beta+1)\log n < \rho < 2\rho$, in c_2 no clock has reached a state with label $EVEN$, and thus, configuration c_2 occurs after configuration c and before configuration c'. Recall that we reach c' from c within $O(n \log n)$ interactions with high probability. In c', some clock has reached position 2ρ, but the other nodes are still either clocks with position in $[\rho, 2\rho)$, or workers in phase 1. Let c'' be a configuration reached after $(3/2)\beta n \log n$ interactions following c'. By Lemma 2.4.9, in c'', all clocks will have positions $\leq 2\rho + 6(\beta+1)\log n < 3\rho$. Combining with the fact that at least one node was at 2ρ in c', maximum gap is $< \rho$, and positions $[\rho, 2\rho)$ have label buffer, we obtain that during the $(3/2)\beta n \log n$ interactions from c' leading to c'', all clocks will be in states with label $EVEN$ or buffer. However, there is at least one clock with label $EVEN$ starting from c', spreading the rumor through workers making them enter phase 2. Due to Lemma 2.4.1, at least half of the nodes are workers. Therefore, by Lemma 2.4.2, in c'', with probability at least $1 - n^{-9}$, all worker nodes are in phase 2. All clocks will be less than gap ρ apart from each other with some clock with a position in $[2\rho, 3\rho)$, and no clock with position $\geq 3\rho$.

We now repeat the argument, but for a duplication phase instead of a cancellation using Lemma 2.4.8, and starting with all clocks with positions in $[2\rho, 3\rho)$ as opposed to $[0, \rho)$ and all workers in phase 2. We consider a sequence of $8(\beta+1)n \log n$ interactions, and by Lemma 2.4.9, no clock will reach position $3\rho + 23(\beta+1)\log n$. Thus, no node will update to an odd phase and since $\delta(c) \geq n/10$, by Lemma 2.4.11, the number of weak nodes must be at least $n/10$ throughout the interaction sequence, allowing the application of Lemma 2.4.8. We get that with high probability, after $O(n \log n)$ rounds, there will again only be clocks and workers in the system. All clocks will be less than gap ρ apart with some clock at a position in $[3\rho, 0)$ and with no clock yet reaching position 0 (wrapping around).

Now, due to the loop structure of the phase clock, we can use the same argument as in Lemma 2.4.10 to claim that, with probability at least $1 - n^{-\beta}$, within $O(n \log n)$ interactions we reach a configuration where some clock is at a position 0 (label *ODD*). Because maximum gap is $< \rho$, all clocks will have label buffer, and the clock at 0 will now spread the rumor making all workers enter phase 3 within the next $(3/2)\beta n \log n$ interactions. No worker will become a terminator, since Lemma 2.4.8 guarantees that all the nodes with value 1 get their values duplicated (turned into 1/2) before they enter phase 3.

Then, we repeat the argument for a cancellation phase (as for phase 1), except that interactions do not create clock nodes (due to *clock-creation=false*) With high probability, within $O(n \log n)$ interactions, all nodes will again be in a worker or a clock state. Moreover, either all strong nodes will support the same decision, or the number of strong nodes supporting each decision will be at most $n/10$. Since by Lemma 2.4.1, the number of clocks is at most $n/2$, δ as defined in Lemma 2.4.11 is at least $n/2 - 2(n/10) - 2(n/10) = n/10$ for this configuration, and will remain so until some node reaches phase 5, allowing us to use Lemma 2.4.8 for phase 4, etc.

Due to Invariant 2.4.4, the case when all strong worker nodes support the same decision must occur before phase $2 \log 1/\epsilon + 1$. Assume that original majority was A, then $Q(c)$ must remain larger than ϵn^2 (up to this point all nodes are clocks or workers, so the condition about D_A holds). The maximum potential in phase $2 \log 1/\epsilon + 1$ is ϵn^2 and it is attained when all nodes are strong and support WIN_A.

Hence, we only need to repeat the argument $O(\log 1/\epsilon)$ times. The number of high probability events that we did union bound over is $O(n \cdot \log 1/\epsilon \cdot \log n)$ (number of interactions for the phase clock). Combining everything, we get that with probability $1 - \frac{O(\log 1/\epsilon)}{n^9}$, the algorithm stabilizes within $O(\log 1/\epsilon \cdot \log n)$ parallel time. □

Lemma 2.4.14. *If the initial majority state has an advantage of ϵn nodes over the minority state, our algorithm stabilizes to the correct majority decision in $O(\log 1/\epsilon \cdot \log n)$ expected parallel time.*

Proof. We know that in the high probability case of Lemma 2.4.13, the protocol stabilizes within $O(\log 1/\epsilon \cdot \log n)$ parallel time. What remains to bound the expectation the low probability events of Lemma 2.4.13.

Notice that as soon as any node gets into an backup or a terminator state, by Lemma 2.4.3 and Lemma 2.4.6, the remaining expected time for the protocol to stabilize is $O(n^2 \log n)$ interactions. Therefore, we will be looking to bound expected time to reach configurations with a backup or a terminator node.

Without loss of generality, suppose A is the inital majority. If all nodes start in A, then the system is already stable with the correct decision. If the initial configuration contains just a single node in state B, then it takes expected $O(n)$ interactions for this node to interact with a node in state A, and lead to a configuration where $n-2$ nodes are in state A (worker state with value 1 and preference WIN_A), one node is a worker with value 0 and one node is a clock with position 0. One of these two nodes (weak worker and the clock) has preference WIN_B and it takes another $O(n)$ expected interactions for it to meet a strong node with preference WIN_A and update its own preference. At that point (after $O(1)$ expected parallel time) the system will be stable with the correct majority decision (since there is only one clock, its position remains at 0, and because of this, workers do not perform any phase updates).

Next, we consider the case when there are at least 2 nodes in state B in the initial configuration. Interactions between two nodes both in state A and two nodes both in state B do not lead to state updates. After one cancellation, as in the previous case, there will be nodes in input states, one clock stuck at position 0, and one weak worker that might change its preference, but not phase or value. Therefore, after $O(n)$ expected interactions, we will get at least two clock nodes in the system.

Unless some node ends up in a backup or a terminator state (this is a good case, as discussed earlier) the number of clocks never decreases. During interactions when there are $k \geq 2$ clocks in the system, the probability of an interaction between two clocks is $\frac{k(k-1)/2}{n(n-1)/2} \geq k/n^2$. Therefore, it takes $O(n^2/k)$ expected interactions for one of the clocks to increment its position. After $k \cdot 4\rho = O(k \log n)$ such increments of some clock position, at least one of the clocks should go through all the possible positions. Notice that this statement is true without the assumption about the maximum gap of the clock (important, because that was a with high probability guarantee, while here we are deriving an expectation bound that holds from all configurations)

Consider any non-clock node v in the system in some configuration c. Since we know how to deal with the case when some node ends up in a backup or a terminator state, suppose v is a worker. The clock node that traverses all positions in $[0, 4\rho)$ necessarily passes through a state with label *ODD* and with label *EVEN*. If v is in an odd phase and does not move to an even phase, then when the clock is in state labelled *EVEN*, there would be $1/n^2$ chance of interacting with v, and vice versa. If such intaraction occurs, and v does not change its state to a non-worker, then it must necessarily increase its phase. Therefore, in any given configuration, for any given worker, the expected number of interactions before it either changes to a non-worker state or increases it phase is $O(k \log n \cdot \frac{n^2}{k} \cdot n^2) = O(n^4 \log n)$.

By Lemma 2.4.1, there can be at most $n/2$ clocks in the system in any configuration. Also, non-worker states can never become worker states again. The maximum number of times a worker can increase its phase is $O(\log n)$. Thus, within $O(n^5 \log^2 n)$ expected interactions, either some node should be in a backup or terminator state, or in the maximum phase possible $(2 \log n + 1)$.

If some worker reaches a maximum phase possible, there are no backup or terminator nodes and there exists another worker with a smaller phase, within $O(n^2)$ expected interactions they will interact. This will either turn both nodes into backups, or the other node will also enter phase $2 \log n + 1$. Thus, within at most $O(n^3)$ additional expected interactions, all workers will be in phase $2 \log n + 1$ (unless there is a backup or a terminator in the system). This contradicts with Invariant 2.4.4, implying that our assumption that no node gets into a backup or a terminator state should be violated within expected $O(n^5 \log^2 n)$ interactions (using linearity of expectation and discarding asymptotically dominated terms). Hence, the protocol always stabilizes within $O(n^4 \log^2 n)$ expected parallel time. The system stabilizes in this expected time in the low probability event of Lemma 2.4.13, giving the total expectated time of at most $O(\log 1/\epsilon \cdot \log n) + \frac{O(\log 1/\epsilon \cdot n^4 \cdot \log^2 n)}{n^9} = O(\log 1/\epsilon \cdot \log n)$ as desired. $\square$

2.5 Synthetic Coin Flips

The state transition rules in population protocols are deterministic, i.e. the interacting nodes do not have access to random coin flips. In this section, we introduce a general technique that extracts randomness from the schedule and after only constant parallel time, allows the interactions to rely on close-to-uniform synthetic coin flips. This turns out to be an useful gadget for designing efficient protocols.

Suppose that every node in the system has a boolean parameter *coin*, initialized with zero. This extra parameter can be maintained independently of the rest of the protocol, and only doubles the state space. When agents x and y interact, they both *flip* the values of their coins. Formally, $x'.coin \leftarrow 1 - x.coin$ and $y'.coin \leftarrow 1 - y.coin$, and the update rule is fully symmetric.

The nodes can use the *coin* value of the interaction partner as a random bit in a randomized algorithm. Clearly, these bits are not independent or uniform. However, we prove that with high probability the distribution of *coin* quickly becomes close to uniform and remains that way. We use the concentration properties of random walks on the hypercube, analyzed previously in other contexts, e.g. [AR16]. We also note that a similar algorithm is used by Laurenti et al. [CKL16] to generate randomness in chemical reaction networks, although they do not prove convergence bounds.

2.5.1 Analysis

We will prove the following guarantee.

Theorem 2.5.1. *For any $i \geq 0$, let X_i be the number of coin values equal to one in the system after i interactions. Fix interaction index $k \geq \alpha n$ for a fixed constant $\alpha \geq 2$. For all sufficiently large n, we have that* $\Pr[|X_k - n/2| \geq n/2^{4\alpha}] \leq 2\exp(-\alpha\sqrt{n}/8)$.

Proof. We label the nodes from 1 to n, and represent their coin values by a binary vector of size n. Let $k_0 = k - \alpha n$, and fix the vector v_0 representing the coin values of the nodes after the interaction of index k_0. For example, if $k_0 = 0$, we know v is a zero vector, because of the way the algorithm is initialized.

For $1 \leq t \leq \alpha n$, denote by Y_t the pair of nodes that are flipped during interaction $k_0 + t$. Then, given v_0 and $Y_1, \ldots, Y_{\alpha n}$, X_k can be computed deterministically. Moreover, it is important to note that Y_j are *independent* random variables and that changing any one Y_j can only change the value of X_k by at most 4. Hence, we can apply McDiarmid's inequality [McD89], stated below.

Claim 2.5.2 (McDiarmid's inequality)**.** *Let $Y_1, \ldots, Y_m$ be independent random variables and let X be a function $X = f(Y_1, \ldots, Y_m)$, such that changing variable Y_j only changes the function value by at most c_j. Then, we have that*

$$\Pr[|X - \mathbb{E}[X]| \geq \epsilon] \leq 2 \cdot \exp\left(-\frac{2\epsilon^2}{\sum_{j=1}^{m} c_j^2}\right).$$

Returning to our argument, assume that the sum of coin values after interaction $k - \alpha n$ is fixed and represented by the vector v_0. In the above inequality, we set $X_k = f_{v_0}(Y_1, \ldots, Y_{\alpha n})$, $\epsilon = \alpha n^{3/4}$ and $c_j = 4$, for all j from 1 to αn. We get that

$$\Pr[|X_k - \mathbb{E}[X_k|] \geq \alpha n^{3/4}] \leq 2 \cdot \exp(-\alpha^2 \sqrt{n}/8).$$

Fixing v_0 also fixes the number of ones among coin values in the system at that moment, which we will denote by x, i.e.

$$x := \sum_{j=1}^{n} v_j(k_0) = X_{k-\alpha n}.$$

We then notice that the following claim holds, whose proof is deferred.

Claim 2.5.3. $\mathbb{E}[X_{i+m} \mid X_i = x] = n/2 + (1 - 4/n)^m \cdot (x - n/2).$

By Claim 2.5.3 we have

$$\mathbb{E}[X_k \mid X_{k-\alpha n} = x] = n/2 + (1 - 4/n)^{\alpha n} \cdot (x - n/2).$$

Since $0 \le x \le n$ and $(1 - 4/n)^{\alpha n} \le \exp(-4\alpha)$, we have that

$$n/2 - n/2^{4\alpha+1} \le \mathbb{E}[X_k \mid X_{k-\alpha n} = x] \le n/2 + n/2^{4\alpha+1}.$$

For any fixed v, we can apply McDiarmid's inequality as above, and get an upper bound on the probability that X_k (given fixed v_0), diverges from the expectation by at most $\alpha n^{3/4}$. But, as we just established, for any v_0, the expectation we get in the bound will be at most $n/2^{4\alpha+1}$ away from $n/2$. Combining these and using that $n/2^{4\alpha+1} \ge \alpha n^{3/4}$ for all sufficiently large n gives the desired bound. □

Next, we provide the missing proof of the claim used above.

Claim 2.5.3. $\mathbb{E}[X_{i+m} \mid X_i = x] = n/2 + (1 - 4/n)^m \cdot (x - n/2)$.

Proof. If two agents both with coin values one are selected, the number of ones decreases by two. If both coin values are zero, it increases by two, and otherwise stays the same. Hence, we have that

$$\begin{aligned}\mathbb{E}[X_{i+m} \mid X_{i+m-1} = t] &= (t-2) \cdot \Pr[X_{i+m} = t-2] \\ &+ t \cdot Pr[X_{i+m} = t] + (t+2) \cdot \Pr[X_{i+m} = t+2] \\ &= (t-2) \cdot \frac{t(t-1)}{n(n-1)} + t \cdot \frac{2t(n-t)}{n(n-1)} \\ &+ (t+2) \cdot \frac{(n-t)(n-t-1)}{n(n-1)} \\ &= t + \frac{2}{n(n-1)} \cdot \left(n^2 - 2nt - n + 2t\right) = t \cdot \left(1 - \frac{4}{n}\right) + 2\end{aligned}$$

Thus, we get a recursive dependence $\mathbb{E}[X_{i+m}] = \mathbb{E}[X_{i+m-1}] \cdot (1 - 4/n) + 2$, that gives

$$\mathbb{E}[X_{i+m}] = 2 \cdot \sum_{j=0}^{m-1} \left(1 - \frac{4}{n}\right)^j + \mathbb{E}[X_i] \cdot \left(1 - \frac{4}{n}\right)^m = \frac{n}{2} + \left(1 - \frac{4}{n}\right)^m \left(x - \frac{n}{2}\right)$$

by telescoping. □

2.5.2 Approximate Counting

Synthetic coins can be used to estimate the number of agents in the system, as follows. Each node executes the coin-flipping protocol, and counts the number of consecutive 1 flips it observes, until the first 0. Each agent records the number of consecutive 1 coin flips as its *estimates*. The agents then exchange their estimates, always adopting the maximum estimate. It is easy to prove that the nodes will eventually stabilize to a number which is a constant-factor approximation of $\log n$, with high probability. This property is made precise in [AAE+17] in the proof of Lemma D.2.

2.6 Phased Leader Election

We partition the state space into *clock* states, *contender* states, and *follower* states. A clock state is just a position on the phase clock loop. A contender state and a follower state share the following two fields (1) a *phase number* in $[1, m]$, which we will fix to $m = O(\log n)$ later, and (2) a $High/Low$ indicator within the phase. Finally, all states have the following bit flags (1) clock-creation, as in the majority protocol, and (2) a coin bit for generating synthetic coin flips with small bias, as described in the previous section. Similar to the phased majority algorithm, we will set $\rho = \Theta(\log n)$ for the phase clock, and have the loop size of the phase clock be $\Psi = 4\rho$. Thus, the state complexity of the algorithm is $\Theta(\log n)$.

Initially, all nodes are contenders, with phase number 1 and a *High* indicator. The coin is initialized with 0 and clock-creation$=$*true*. Each node flips its coin at every interaction. As in the majority, we assign one of three different labels to each clock position, *buffer*, *ODD* and *EVEN*. Only contenders map to the leader output, and our goal is stabilize to a single contender.

Clock States and Flags: Clock nodes, as in Section 2.4, implement the phase clock from Section 2.3 to update their position. When a clock with clock-creation$=$*true* reaches the threshold T_c, it sets clock-creation to *false*. The threshold is set as in the phased majority in Section 2.4, and clock-creation flag works exactly the same way.

Contenders and Followers: The general idea of the followers comes from [AG15], and it is also used in [AAE+17]. They help contenders eliminate each other by transmitting information.

More precisely, a follower maintains a maximum pair of a phase number and a *High*/*Low* indicator (lexicographically ordered, *High* $>$ *Low*) ever encountered in an interaction partner (regardless of whether the partner was a contender or a follower). When a contender meets another contender or a follower with a larger phase-indicator pair than its own, it becomes a follower and adopts the pair. A contender or a follower with a strictly larger pair than its interaction partner does not update its state. Also, when a contender and a follower meet and they both have the same pair, both remain in their respective states.

When two contenders with the same pair interact and clock-creation$=$*true*, one of them becomes a clock at position 0. If clock-creation$=$*false*, then one of them becomes a follower with the same pair. The other contender remains in the same state. For technical reasons, we want to avoid creating more than $n/2$ clocks. This can be accomplished by adding a single *created* bit initialized to 0. When two contenders with the same pair meet, and both of their *created* bit is 0, then one of them becomes a clock and another sets *created* to 1. Otherwise, if one of the contenders has *created*$= 1$, then it becomes a follower; the other remains unchanged. Then Lemma 2.4.1 still works and gives that we will never have more than $n/2$ clocks.

Contender Phase Update Consider a contender in phase ϕ. If ϕ is odd phase and the contender meets a clock whose state has an *EVEN* label, or when ϕ is even and the contender meets a clock with an *ODD*-labelled state, then it increments its phase number to $\phi + 1$. However, again due to technical reasons (to guarantee unbiased synthetic randomness), entering the next phase happens in two steps. First the node changes to a special *intermediate* state (this can be implemented by a single bit that is true if the state is intermediate), and only after the next interaction changes to non-intermediate contender with phase $\phi + 1$ and sets the *High*/*Low* indicator to the coin value of the latest interaction partner. If the coin was 1, indicator is set to

High and if the coin was 0, then it is set to *Low*. For the partner, meeting with an intermediate state is almost like missing an interaction - only the coin value is flipped. An exception to the rule of incrementing the phase is obviously when a contender is in phase m. Then the state does not change.

2.6.1 Analysis

Here we show how to adjust analysis of the phased majority algorithm to get the desired guaranteed for phased leader election.

Theorem 2.6.1. *Our algorithm elects a unique stable leader within $O(\log^2 n)$ parallel time, both with high probability and in expectation.*

Proof. We first prove that is always at least one contender in the system. Assume the contrary, and consider the interaction sequence leading to a contenderless configuration. Consider the contender which had the highest phase-indicator pair when it got eliminated, breaking ties in favor of the later interaction. This is a contradiction, because no follower or other contender may have eliminated it, as this requires having a contender with a larger phase-indicator pair.

By construction, the *interacted* bit combined with Lemma 2.4.1 ensures that there are never more than $n/2$ clocks in the system. We set up the phase clock with the same ρ as in majority, and also the clock-creation threshold $T_c = 23(\beta + 1) \log n$. After the first $8(\beta + 1)n \log n$ interactions, with probability $1 - n^{-\beta}$, there will be at least $2/5n$ clocks. The proof of this claim is similar to Lemma 2.4.8: if the number of contenders with initial state and created set to 0 was at least $n/10$ throughout the sequence of interactions, then any given node would have interacted with such node with high probabiliy, increasing the number of clocks. Otherwise, the number of nodes with *created* $= 0$ falls under $n/10$, but there are as many nodes that are clocks as contenders that are not *created* $= 0$ and at least $(n - n/10)/2 > 2n/5$.

Now we can apply the same argument as in Lemma 2.4.13 and get that, with high probability, the nodes will keep entering larger and larger phases. In each phase, as in the majority argument, a rumor started at each node reaches all other nodes with high probability. This means that if a contender in a phase selects indicator *High*, then all other contenders that select indicator *Low* in the same phase will get eliminated with high probability. By Theorem 4.1 from [AAE+17], the probability that a given contender picks *High* is at least $1/2 - 1/2^8$ with probability at least $1 - 2\exp(-\sqrt{n}/4)$. For every other node, the probability of choosing *Low* is similarly lower bounded. Thus, Markov's inequality implies that in each phase, the number of contenders decreases by a constant fraction with constant probability, and phases are independent of each other. By a Chernoff bound, it is sufficient to take logarithmically many phases to guarantee that one contender will remain, with high probability, taking a union bound with the event that each phase takes $O(\log n)$ parallel time, as proved in Lemma 2.4.13

For the expected bound, observe that when there are more than two contenders in the system, there is $1/n^2$ probability of their meeting. Hence, the protocol stabilizes from any configuration, in particular in the with low probability event, within $O(n^3)$ interactions, which does not affect the total expected parallel time of $O(\log^2 n)$. $\square$

2.7 Lower Bounds

In the previous sections, we have been implicitly consistent with notation in the following sense: we have primarily used capital latin letters for states, small latin letters for configurations, and greek letters for constants. We may not abide with these principles in this section, because we will instead be consistent with the notation used in the lower bound papers, and also with the original notation of the technical tools that we use and extend [Dot14, CCDS15, DS15, AAE+17].

We start by the developing this technical machinery.

2.7.1 Technical Tools

Fix a function $f : \mathbb{N} \to \mathbb{R}^+$. Consider a configuration c reached by an execution of a protocol $\mathcal{P}_k$ (a protocol with k states), and states $r_1, r_2 \in \Lambda_k$. A transition $\alpha : (r_1, r_2) \to (z_1, z_2)$ is an *f-bottleneck* for c, if $c(r_1) \cdot c(r_2) \leq f(|c|)$. This bottleneck transition implies that the probability of a transition $(r_1, r_2) \to (z_1, z_2)$ is bounded. Hence, proving that transition sequences from initial configuration to final configurations contain a bottleneck implies a lower bound on the stabilization time. Conversely, if a protocol stabilizes fast, then it must be possible to stabilize using a transition sequence which does not contain any bottleneck.

Given a protocol $\mathcal{P}_k$ with k states executing in a system of n agents, for a configuration c and a set of configurations Y, let us define $T[c \Longrightarrow Y]$ as the expected parallel time it takes from c to reach some configuration in Y for the first time.

Lemma 2.7.1. *In a system of n nodes executing protocol $\mathcal{P}_k$, let $f : \mathbb{N} \to \mathbb{R}^+$ be a fixed function, $c : \Lambda_k \to \mathbb{N}$ be a configuration, and Y be a set of configurations, such that every transition sequence from c to some $y \in Y$ has an f-bottleneck. Then it holds that $T[c \Longrightarrow Y] \geq \frac{n-1}{2f(n)k^2}$.*

Proof. By definition, every transition sequence from c to a configuration $y \in Y$ contains an f-bottleneck, so it is sufficient to lower bound the expected time for the first f-bottleneck transition to occur from c before reaching Y. In any configuration c' reachable from c, for any pair of states $r_1, r_2 \in \Lambda_k$ such that $(r_1, r_2) \to (o_1, o_2)$ is an f-bottleneck transition in c', the definition implies that $c'(r_1) \cdot c'(r_2) \leq f(n)$. Thus, the probability that the next pair of agents selected to interact are in states r_1 and r_2, is at most $\frac{2f(n)}{n(n-1)}$. Taking an union bound over all k^2 possible such transitions, the probability that the next transition is f-bottleneck is at most $k^2 \frac{2f(n)}{n(n-1)}$. Bounding by a Bernoulli trial with success probability $\frac{2f(n)k^2}{n(n-1)}$, the expected number of interactions until the first f-bottleneck transition is at least $\frac{n(n-1)}{2f(n)k^2}$. The expected parallel time is this quantity divided by n, completing the argument. □

Lemma 2.7.2. *Consider a population protocol $\mathcal{P}_k$ for majority, executing in a system of n agents. Fix a function f. Assume that $\mathcal{P}_k$ stabilizes in expected time $o\left(\frac{n}{f(n) \cdot k^2}\right)$ from an initial configuration i_n. Then, for all sufficiently large n, there exists a configuration y_n with n agents and a transition sequence q_n, such that (1) $i_n \Longrightarrow_{q_n} y_n$, (2) q_n has no f-bottleneck, and (3) y_n has a stable majority decision.*

Proof. We know that the expected stabilization time from i_n is finite. Therefore, a configuration y_n that has a stable majority decision must be reachable from i_n through some transition sequence q_n. However, we also need q_n to satisfy the second requirement.

Let Y_n be a set of all stable output configurations with n agents. Suppose for contradiction that every transition sequence from i_n to some $y \in Y_n$ has an f-bottleneck. Then, using Lemma 2.7.1, the expected time to stabilize from i_n to a majority decision is $T[i_n \implies Y_n] \geq \frac{n-1}{2f(n)k^2} = \Theta(\frac{n}{f(n)k^2})$. But we know that the protocol stabilizes from i_n in time $o(\frac{n}{f(n)k^2})$, and the contradiction completes the proof. $\square$

For our general state lower bounds, we will also develop a slightly different version of the above lemma. It is based on the fact that all protocols with a limited state count will end up in a configuration in which all states are present in large counts. The following statement, provided here without a proof, generalizes the main result of [Dot14] to a super-constant state space. We prove it in [AAE+17], Lemma A.1.

Lemma 2.7.3 (Density Lemma). *Consider a population protocol $\mathcal{P}_k$ executing in a system of n agents, where $k \leq 1/2 \log\log n$, from an initial configuration i_n, where for each state $s \in \Lambda_k$, either $i_n(s) = 0$, or $i_n(s) \geq n/M$, for some constant M. Without loss of generality, assume that every state in Λ_k can actually be produced by some transition sequence*[5]*. Then, with probability $\geq 1 - (1/n)^{0.99}$, the execution will reach some configuration c, such that for every state $s \in \Lambda_k$, we have $c(s) \geq n^{0.99}$.*

Here is the promised modified version of Lemma 2.7.2.

Lemma 2.7.4. *Consider a population protocol $\mathcal{P}_k$ executing in a system of n agents, where $k \leq 1/2 \log\log n$, from an initial configuration i_n, where for each state $s \in \Lambda_k$, either $i_n(s) = 0$, or $i_n(s) \geq n/M$, for some constant M. Without loss of generality, assume that every state in Λ_k can be produced by some transition sequence. Fix a function f. Assume that $\mathcal{P}_k$ stabilizes in expected time $o\left(\frac{n}{f(n) \cdot k^2}\right)$ from i_n. Then, for all sufficiently large n, there exist configuration x_n, y_n with n agents, and a transition*

[5]Otherwise, we would just consider the subset of the states in Λ_k that are actually producible.

sequence q_n, such that: (1) $i_n \Longrightarrow x_n$, (2) $x_n(s) \geq n^{0.99}$ for all $s \in \Lambda_k$, (3) $x_n \Longrightarrow_{q_n} y_n$, (4) q_n has no f-bottleneck, and (5) y_n is a stable output configuration.

Proof. The proof is analogous to the proof of Lemma 2.7.2, except we first reach configuration x_n from i_n using the Density Lemma, and then reach y_n from x_n instead of from i_n. The density lemma ensures that x_n satisfies the first two properties. Moreover, since the probability of reaching a configuration that can serve as x_n is $1 - (1/n)^{0.99}$, the stabilization speed of $\mathcal{P}_k$ from x_n must also be $o\left(\frac{n}{f(n)\cdot k^2}\right)$, allowing us to apply the proof argument of Lemma 2.7.2.

A detailed proof of this statement can also be found in [AAE+17], Lemma A.3. $\square$

Hence, fast stabilization requires the existence of a bottleneck-free transition sequence. The next *transition ordering lemma*, proves a property of such a transition sequence: there exists an order over all states whose counts decrease by a large margin such that, for each of these states d_j, the sequence contains at least a certain number of a specific transition that consumes d_j, but does not consume or produce any states $d_1, \ldots, d_{j-1}$ that are earlier in the ordering. This lemma is due to [CCDS15], and with minor modifications plays major role in the lower bounds of [DS15, AAE+17]. The proof is omitted and can be found in any of these references (the formulation with f-bottleneck is proved in [AAE+17], Lemma A.4)

Lemma 2.7.5 (Transition Ordering Lemma). *Consider a population protocol $\mathcal{P}_k$, executing in a system of n agents. Fix $b \in \mathbb{N}$, and let $\beta = k^2b + kb$. Let $x, y : \Lambda_k \to \mathbb{N}$ be configurations of n agents, such that for all states $s \in \Lambda_k$ we have $x(s) \geq \beta^2$ and $x \Longrightarrow_q y$ via a transition sequence q without a β^2-bottleneck. Define*

$$\Delta = \{d \in \Lambda_k \mid y(d) \leq b\}$$

to be the set of states whose count in configuration y is at most b. Then there is an order $\Delta = \{d_1, d_2, \ldots, d_m\}$, such that, for all $j \in \{1, \ldots, m\}$, there is a transition α_j of the form $(d_j, s_j) \to (o_j, o'_j)$ with $s_j, o_j, o'_j \notin \{d_1, \ldots, d_j\}$, and α_j occurs at least b times in q.

For protocols that use $\Omega(\log\log n)$ states, *dense* intermediate configurations may no longer occur. Instead, we prove the following *suffix transition ordering lemma*, which considers the suffix of the ordering starting with some state whose count decreases by a large margin.

Lemma 2.7.6 (Suffix Transition Ordering Lemma)**.** *Let $\mathcal{P}_k$ be a population protocol executing in a system of n agents. Fix $b \in \mathbb{N}$, and let $\beta = k^2 b + kb$. Let $x, y : \Lambda_k \to \mathbb{N}$ be configurations of n agents such that for a state $A' \in \Lambda_k$ (1) $x \Longrightarrow_q y$ via a transition sequence q without a β^2-bottleneck. (2) $x(A') \geq \beta$, and (3) $y(A') = 0$. Define*

$$\Delta = \{d \in \Lambda_k \mid y(d) \leq b\}$$

to be the set of states whose count in configuration y is at most b. Then there is an order $\{d_1, d_2, \ldots, d_m\}$, such that $d_1 = A'$ and for all $j \in \{1, \ldots, m\}$ (1) $d_j \in \Delta$, and (2) there is a transition α_j of the form $(d_j, s_j) \to (o_j, o'_j)$ that occurs at least b times in q. Moreover, $s_j, o_j, o'_j \in (\Lambda_k - \Delta) \cup \{d_{j+1}, \ldots, d_m\}$.

Proof. We know by definition that $A' \in \Delta$. We will construct the ordering in reverse, i.e. we will determine e_j for $j = |\Delta|, |\Delta| - 1, \ldots$ in this order, until $e_j = A'$. Then, we set $m = |\Delta| - j + 1$ and $d_1 = e_j, \ldots, d_m = e_{|\Delta|}$.

We start by setting $j = |\Delta|$. Let $\Delta_{|\Delta|} = \Delta$. At each step, we will define the next Δ_{j-1} as $\Delta_j - \{e_j\}$. We define $\Phi_j : (\Lambda_k \to \mathbb{N}) \to \mathbb{N}$ based on Δ_j as $\Phi_j(c) = \sum_{d \in \Delta_j} c(d)$, i.e. the number of agents in states from Δ_j in configuration c. Notice that once Δ_j is well-defined, so is Φ_j.

The following works for all j as long as $e_{j'} \neq A'$ for all $j' > j$, and thus, lets us construct the ordering. Because $y(d) \leq b$ for all states in Δ, it follows that $\Phi_j(y) \leq jb \leq kb$. On the other hand, we know that $x(A') \geq \beta$ and $A' \in \Delta_j$, so $\Phi_j(x) \geq \beta \geq kb \geq \Phi_j(y)$. Let c' be the last configuration along q from x to y where $\Phi_j(c') \geq \beta$, and r be the suffix of q after c'. Then, r must contain a subsequence of transitions u each of which strictly decreases Φ_j, with the total decrease over all of u being at least $\Phi_j(c') - \Phi_j(y) \geq \beta - kb \geq k^2 b$.

Let $\alpha : (r_1, r_2) \to (p_1, p_2)$ be any transition in u. α is in u so it strictly decreases Φ_j, and without loss of generality $r_1 \in \Delta_j$. Transition α is not a β^2-bottleneck since q does not contain such bottlenecks, and all configurations c along u have $c(d) < \beta$ for all $d \in \Delta_j$ by definition of r. Hence, we must have $c(r_2) > \beta$ meaning $r_2 \notin \Delta_j$. Exactly one state in Δ_j decreases its count in transition α, but α strictly decreases Φ_j, so it must be that both $p_1 \notin \Delta_j$ and $p_2 \notin \Delta_j$. We take $d_j = r_1, s_j = r_2, o_j = p_1$ and $o'_j = p_2$.

There are k^2 different types of transitions. Each transition in u decreases Φ_j by one and there are at least $k^2 b$ such instances, at least one transition type must repeat in u at least b times, completing the proof. $\square$

The next lemma for majority protocols that satisfy output dominance lets us apply the Suffix Transition Ordering Lemma, with A' set to the initial minority state.

Lemma 2.7.7. *Let $\mathcal{P}$ be a monotonic population protocol satisfying output dominance that stably computes majority decision for all sufficiently large n using $s(n, \epsilon)$ states. For all sufficiently large n, consider executing protocol $\mathcal{P}_{s(n,\epsilon)}$ in a system of $n' < n/2$ agents, from an initial configuration $i_{n'}$ with $\epsilon n'$ more agents in state B. Consider any c with $i_{n'} \Longrightarrow c$, that has a stable majority decision WIN_B. Then $c(A) = 0$.*

Proof. Notice that for sufficiently large n, we can consider executing protocol $\mathcal{P}_{s(n,\epsilon)}$ from an initial configuration $i_{n'}$, and know that it stabilizes to the correct majority decision, because $\mathcal{P}$ is a monotonic protocol.

Assume for contradiction that $c(A) > 0$. Since c has a stable majority decision WIN_B, we must have $\gamma_{s(n,\epsilon)}(A) = WIN_B$. Now consider a system of n agents, executing $\mathcal{P}_{s(n,\epsilon)}$, where n' agents start in configuration $i_{n'}$ and reach c, and the remaining agents each start in state A. Clearly, for the system of $n > 2n'$ agents, A is the majority. Define c' to be configuration c plus $n - n'$ agents in state A. We only added agents in state A from c to c' and $c(A) > 0$, thus for any state $s \in \Lambda_{s(n,\epsilon)}$ with $c'(s) > 0$, we have $c(s) > 0$. However, as c has a stable majority WIN_B, by output closedness, any configuration c'' with $c' \Longrightarrow c''$ that has a stable majority decision, should have a decision WIN_B.

As $\mathcal{P}$ stably computes the majority decision, $\mathcal{P}_{s(n,\epsilon)}$ should stabilize in a finite expected time for n agents. c' is reachable from an initial configuration of n agents. Thus, some configuration c'' with a stable majority decision must be reachable from c'. However, the initial configuration has majority A, and c'' has a majority decision WIN_B, a contradiction. □

2.7.2 Output-Dominant Majority

In this section, we prove the following.

Theorem 2.7.8. *Assume any monotonic population protocol $\mathcal{P}$ satisfying output dominance, which stably computes majority decision using $s(n,\epsilon)$ states. Then, the time complexity of $\mathcal{P}$ must be $\Omega\left(\frac{n-2\epsilon n}{3^{2s(n,\epsilon)} \cdot s(n,\epsilon)^7 \cdot (\epsilon n)^2}\right)$.*

Proof. We will proceed by contradiction. Assume a protocol $\mathcal{P}_{s(n,\epsilon)}$ which would contradict the lower bound.

Then, for all sufficiently large n, $\mathcal{P}_{s(n,\epsilon)}$ stably computes majority decision in expected parallel time $o\left(\frac{n-2\epsilon n}{3^{2 \cdot s(n,\epsilon)} \cdot s(n,\epsilon)^7 \cdot (\epsilon n)^2}\right)$. We denote $k = s(n,\epsilon)$, $n' = \frac{n-2\epsilon n}{k+1}$, $b(n) = 3^k \cdot (2\epsilon n)$ and $\beta(n) = k^2 \cdot b(n) + k \cdot b(n)$. Let $i_{n'}$ be an initial configuration of n' agents, with $\epsilon n'$ more agents in state B.

By monotonicity of the protocol $\mathcal{P}$, $\mathcal{P}_k$ should also stabilize from $i_{n'}$ in expected time $o\left(\frac{n-2\epsilon n}{3^{2k} \cdot k^7 \cdot (\epsilon n)^2}\right)$, which is the same as $o\left(\frac{n'}{k^2 \cdot \beta(n)^2}\right)$. Thus, by Lemma 2.7.2, there exists a transition sequence q without a $\beta(n)^2$ bottleneck, and configuration $y_{n'}$ with a stable majority decision, such that $i_{n'} \Longrightarrow_q y_{n'}$.

The bound is only non-trivial in a regime where $\epsilon n \in o(\sqrt{n})$, and $n' = \frac{n-2\epsilon n}{k+1} \in \omega(k^2 \cdot \beta(n)^2)$. In this regime, we have $i_{n'}(A) = \frac{n'-\epsilon n'}{2} \geq \beta(n)$ for all sufficiently large n. Also, by Lemma 2.7.7, $y_{n'}(A) = 0$. Therefore, we can apply the suffix transition ordering Lemma 2.7.6 with $\mathcal{P}_k$, $b = b(n)$ and $\beta = \beta(n)$. This gives an ordering $\{d_1, \ldots, d_m\}$ on a subset of Δ and corresponding transitions α_j.

Claim 2.7.9. *Let $n'' = n' \cdot (m+1) + 2\epsilon n$ and i be an initial configuration of n'' agents consisting of $m+1$ copies of configuration $i_{n'}$ plus $2\epsilon n$ agents in state A. Then,*

$i \Longrightarrow z$, for a configuration z, such that for all $s \in \Lambda_k$, if $z(s) > 0$ then $y_{n'}(s) > 0$.

We prove the claim later in this section, right after the main theorem.

Returning to the main thread, we have $n'' \leq n$ due to $m \leq k$. Moreover, the initial configuration i of n'' agents has at least $\epsilon n \geq \epsilon n''$ more agents in state A than B (since $(m+1) \cdot \epsilon n' \leq \epsilon n$, which follows from $(m+1)n' \leq (k+1)n' \leq n$). So, monotonicity of $\mathcal{P}$ implies that $\mathcal{P}_k$ also stably computes majority decision from initial configuration i. We know $i \Longrightarrow z$, so it must be possible to reach a configuration y from z that has a stable majority decision (otherwise $\mathcal{P}_k$ would not have a finite time complexity to stabilize from i). By output dominance property of $\mathcal{P}$ for z and $y_{n'}$, y has to have the same majority decision as $y_{n'}$. However, the correct majority decision is WIN_B in $i_{n'}$ and WIN_A in i. □

Now we prove the inductive claim.

Claim 2.7.9 (Surgery)**.** *Let $n'' = n' \cdot (m+1) + 2\epsilon n$ and i be an initial configuration of n'' agents consisting of $m+1$ copies of configuration $i_{n'}$ plus $2\epsilon n$ agents in state A. Then, $i \Longrightarrow z$, for a configuration z, such that for all $s \in \Lambda_k$, if $z(s) > 0$ then $y_{n'}(s) > 0$.*

Proof. In this proof, we consider transition sequences that might temporarily bring counts of agents in certain states below zero. This will not be a problem because later we add more agents in these states, so that the final transition sequence is well-formed, meaning that no count ever falls below zero.

We do the following induction. For every j with $1 \leq j \leq m$, consider an initial configuration ι_j consisting of j copies of configuration $i_{n'}$ plus $2\epsilon n$ agents in state A. Then, there exists a transition sequence q_j from ι_j that leads to a configuration z_j, with the following properties:

1. For any $d \in \Delta - \{d_{j+1}, \ldots, d_m\}$, the count of agents in d remains non-negative throughout q_j. Moreover, if $y_{n'}(d) = 0$, then $z_j(d) = 0$.

2. For any $d \notin \Delta - \{d_{j+1}, \ldots, d_m\}$ the minimum count of agents in d during q_j is $\geq -3^j \cdot (2\epsilon n)$.

3. For any $d \in \{d_{j+1}, \dots, d_m\}$, if $y_{n'}(d) = 0$, then $|z_j(d)| \leq 3^j \cdot (2\epsilon n)$.

The base case: Consider $j = 1$. Here ι_1 is simply $i_{n'}$ combined with $2\epsilon n$ agents in state A. We know $i_{n'} \Longrightarrow_q y_{n'}$. Thus, from ι_1 by the same transition sequence q we reach a configuration $y_{n'}$ plus $2\epsilon n$ agents in state $d_1 = A$. Moreover, by suffix transition ordering lemma, we know that transition α_1 of form $(A, s_1) \rightarrow (o_1, o_1')$ occurs at least $b(n) \geq (2\epsilon n)$ times in q. We add $2\epsilon n$ occurences of transition α_1 at the end of q and let q_1 be the resulting transition sequence. z_1 is the configuration reached by q_1 from ι_1.

For any $d \in \Lambda_k$, during the transition sequence q, the counts of agents are non-negative. In the configuration after q, the count of agents in state $d_1 = A$ is $y_{n'}(A) + 2\epsilon n = 2\epsilon n$, and during the remaining transitions of q_1 ($2\epsilon n$ occurences of α_1), the count of agents in A remains non-negative and reaches $z_1(d_1) = 0$ as required (since $y_{n'}(d_1) = y_{n'}(A) = 0$). $s_1, o_1, o_1' \in (\Lambda_k - \Delta) \cup \{d_2, \dots d_m\}$ implies that for any state $d \in \Delta - \{d_1, d_2, \dots, d_m\}$, the count of agents in d remains unchanged and non-negative for the rest of q_1. Moreover, $z_1(d) = y_{n'}(d)$, thus if $y_{n'}(d) = 0$ then $z_1(d) = 0$. This completes the proof of the first property.

Now, consider any $d \notin \Delta - \{d_2, \dots, d_m\}$. The count of d is non-negative during q, and might decrease by at most $2\epsilon n < 3^j \cdot (2\epsilon n)$ during the remaining $2\epsilon n$ occurences of transition α_1 in q_1 (achieved only when $s_1 = d$ and $s_1 \neq o_1, o_1'$). This proves the second property.

The final count of any state in z_1 differs by at most $2 \cdot (2\epsilon n) \leq 3^j \cdot (2\epsilon n)$ from the count of the same state in $y_{n'}$. (the only states with different counts can be s_1, o_1 and o_1', and the largest possible difference of precisely $2 \cdot (2\epsilon n)$ is attained when $o_1 = o_1'$). This implies the third property.

Inductive step: We assume the inductive hypothesis for some $j < m$ and prove it for $j+1$. Inductive hypothesis gives us configuration ι_j and a transition sequence q_j to another configuration z_j, satisfying the three properties for j. We have $\iota_{j+1} = i_{n'} + \iota_j$, adding another new configuration $i_{n'}$ to previous ι_j.

Let u be the minimum count of state d_{j+1} during q_j. If $u \geq 0$, we let $q^1_{j+1} = q$. Otherwise, we remove $|u| \leq 3^j \cdot (2\epsilon n) \leq b(n)$ instances of transition α_{j+1} from q, and call the resulting transition sequence q^1_{j+1}.

Now from $\iota_{j+1} = i_{n'} + \iota_j$ consider performing transition sequence q^1_{j+1} followed by q_j. q^1_{j+1} affects the extra configuration $i_{n'}$ (difference between ι_j and ι_{j+1}), and produces $|u|$ extra nodes in state d_{j+1} if u was negative. Now, when q_j is performed afterwards, the count of state d_{j+1} never becomes negative.

Let v be the count of d_{j+1} in the configuration reached by the transition sequence q^1_{j+1} followed by q_j from ι_{j+1}. Since the count never becomes negative, we have $v \geq 0$. If $y_{n'}(d_{j+1}) > 0$, then we let this sequence be q_{j+1}. If $y_{n'}(d_{j+1}) = 0$, then we add v occurences of transition α_{j+1}, i.e. q_{j+1} is q^1_{j+1} followed by q_j followed by v times α_{j+1}. The configuration reached from ι_{j+1} by q_{j+1} is z_{j+1}.

Consider $d \in \Delta - \{d_{j+2}, \dots, d_m\}$. For $d = d_{j+1}$, if $y_{n'}(d_{j+1}) = 0$, then we ensured that $z_{j+1}(d_{j+1}) = 0$ by adding v occurences of transitions α_{j+1} at the end. In fact, by construction, the count of agents in d_{j+1} never becomes negative during q_{j+1}. It does not become negative during q^1_{j+1} and the $|u|$ extra nodes in state d_{j+1} that are introduced ensure futher non-negativity of the count during q_j. Finally, if the count is positive and $y_{n'}(d_{j+1}) = 0$, it will be reduced to 0 by the additional occurences of transition α_{j+1}, but it will not become negative. For $d \in \Delta - \{d_{j+1}, d_{j+2}, \dots, d_m\}$, recall that $\alpha_{j+1} = (d_{j+1}, s_{j+1}) \to (o_{j+1}, o'_{j+1})$, where $s_{j+1}, o_{j+1}, o'_{j+1} \in (\Lambda_k - \Delta) \cup \{d_{j+2}, \dots d_m\}$. Thus, none of $s_{j+1}, o_{j+1}, o'_{j+1}$ are equal to d. This implies that the count of agents in d remain non-negative during q_{j+1} as the removal and addition of α_{j+1} does not affect the count (count is otherwise non-negative during q; also during q_j by inductive hypothesis). If $y_{n'}(d) = 0$, we have $z_{j+1}(d) = z_j(d) + y_{n'}(d) = 0$, as desired. This proves the first property.

The states for which the minimum count of agents during q_{j+1} might be smaller than during q_j are s_{j+1}, o_{j+1} and o'_{j+1}. Let us first consider o_{j+1} and o'_{j+1}. In our construction, we might have removed at most $3^j \cdot (2\epsilon n)$ occurences of α_{j+1} from q to get q^1_{j+1}, and the largest decrease of count would happen by $2 \cdot 3^j \cdot (2\epsilon n)$ if $o_{j+1} = o'_{j+1}$. Adding transitions α_{j+1} at the end only increases the count of o_{j+1} and

o'_{j+1}. Therefore, the minimum count of agents for these two states is $-3^j \cdot (2\epsilon n) - 2 \cdot 3^j \cdot (2\epsilon n) = -3^{j+1} \cdot (2\epsilon n)$, as desired. Now consider state s_{j+1}. We can assume $s_{j+1} \neq o_{j+1}, o'_{j+1}$ as otherwise, the counts would either not change or can be analyzed as above for o_{j+1}. Removing occurences of transition α_{j+1} only increases count of s_{j+1}, and it only decreases if we add v occurences of α_{j+1} at the end to get the count of d_{j+1} to 0. Since $y_{n'}(d_{j+1})$ should be 0 in this case in order for us to add transitions at the end, we know $v = z_j(d_{j+1})$ if $u \geq 0$, and $v = z_j(d_{j+1}) + |u|$ if $u < 0$. In the second case, we remove $|u|$ occurences before adding v occurences, so the minimum count in both cases decreases by at most $|z_j(d_{j+1})|$. By induction hypothesis the minimum count is $\geq -3^j \cdot (2\epsilon n)$ and $|z_j(d_{j+1})| \leq 3^j \cdot (2\epsilon n)$, so the new minimum count of s_{j+1} is $\geq -2 \cdot 3^j \cdot (2\epsilon n) \geq -3^{j+1} \cdot (2\epsilon n)$. This proves the second property.

In order to bound the maximum new $|z_{j+1}(d)|$ for $d \in \{d_{j+2}, \ldots, d_m\}$ with $y_{n'}(d) = 0$, we take a similar approach. Since $y_{n'}(d) = 0$, if $|z_{j+1}(d)|$ differs from $|z_j(d)|$, then d must be either s_{j+1}, o_{j+1} or o'_{j+1}. The minimum negative value that $z_{j+1}(d)$ can achieve can be shown to be $3^{j+1} \cdot (2\epsilon n)$ with the same argument as in the previous paragraph - considering $d = o_{j+1} = o'_{j+1}$ and $d = s_{j+1}$ and estimating the maximum possible decrease, combined with $|z_j(d)| \leq 3^j \cdot (2\epsilon n)$. Let us now bound the maximum positive value. If $d = o_{j+1} = o'_{j+1}$, the increase caused by v additional occurences of α_{j+1} at the end of q_{j+1} is $2v$. As before, $v = z_j(d_{j+1})$ if $u \geq 0$, and $v = z_j(d_{j+1}) + |u|$ if $u < 0$, and in the second case, we also decrease the count of d by $2|u|$ when removing $|u|$ occurences of α_{j+1} to build q^1_{j+1} from q. Thus, the maximum increase is $2|z_j(d_{j+1})| \leq 2 \cdot 3^j \cdot (2\epsilon n)$. If $d = s_{j+1}$, then the only increase comes from at most $|u| \leq 3^j \cdot (2\epsilon n)$ removed occurences of α_{j+1}. Therefore, the maximum positive value of $z_{j+1}(d)$ equals maximum positive value of $z_j(d)$ which is $3^j \cdot (2\epsilon n)$ plus the maximum possible increase of $2 \cdot 3^j \cdot (2\epsilon n)$, giving $3^{j+1} \cdot (2\epsilon n)$ as desired. This completes the proof for the third property and of the induction.

The rest of the proof: We take $i = i_{n'} + \iota_m$ and $z = y_{n'} + z_m$. The transition sequence p from i to z starts by q from $i_{n'}$ to $y_{n'}$, followed by q_m.

By the first property of q_m, and the fact that no count is ever negative in q from

$i_{n'}$ to $y_{n'}$, for any $d \in \Delta$, the count of agents in state d never becomes negative during p. Next, consider any state $d \in \Lambda_k - \Delta$. By the second property, when q_m is executed from ι_m to z_m, the minimum possible count in q_m is $-3^m \cdot (2\epsilon n)$. However, in transition sequence p, q_m from ι_m to z_m follows q, and after q we have an extra configuration $y_{n'}$ in the system. By the definition of Δ, $y_{n'}(d) \geq b(n) \geq 3^k \cdot (2\epsilon n) \geq 3^m \cdot (2\epsilon n)$. Therefore, the count of agents in d also never becomes negative during p, and thus the final transition sequence p is well-formed.

Now, consider a state s, such that $y_{n'}(s) = 0$. We only need to show that $z(s) = 0$. By definition of Δ, we have $s \in \Delta$, and the first property implies $z(s) = z_m(s) = 0$, completing the proof. $\square$

This lower bound implies, for instance, that for $\epsilon = 1/n$, a monotonic protocol satisfying output dominance and stably solves majority using $\log n/(4\log 3)$ states, needs to have time complexity $\Omega(\sqrt{n}/\mathsf{polylog} n)$.

Weak Monotonicity:

We use monotonicity of the protocol to invoke the same protocol with different number of nodes. In particular, in Theorem 2.7.8, if the protocol uses k states for n nodes, we need to be able to use the same protocol for n/k nodes. Suppose instead that the protocol is weakly monotonic. If the state complexity is $k \leq \log n/(2\log\log n)$, then we can find infinitely many n with the desired property that the same protocol works for n/k and n nodes. This allows us to apply the same lower bound argument, but we would only get a lower bound for state complexities up to $\log n/(2\log\log n)$.

Non-Dense Initial Configurations:

Unlike [DS15, AAE$^+$17], we do not (and cannot) require fast stabilization from configurations where all states have large counts, which necessitated starting in "dense" initial configurations. Our lower bound works from more general initial configurations. For instance, suppose that $\theta(n)$ nodes start in states A and B, among which we must compute the majority decision, but the remaining nodes can be in arbitrary states. Even if a protocol is only required to stabilize fast when a single leader is provided in the initial configuration, our lower bound applies. Without modification, the lower bound argument does require that the protocol should not stabilize to a wrong decision even if initially there are multiple nodes in this designated leader state. However, we can assume that the leader states do not need to map to any output. Thus, the lower bound applies for instance to protocols where all non-leader nodes would eventually stabilize to the correct output, and stabilize fast only if there was a single leader to start with.

2.7.3 General Lower Bound

In this section, we prove a weaker lower bound for leader election and majority, but only under the weak monotonicity assumption.

Our framework here is based on [DS15], but differs from it in a few key points. Specifically, the crux of the argument in [DS15] is the existence of a set Γ of *unbounded states*, whose counts grow unbounded as the number of agents n tends to infinity. The unbounded property of Γ is used multiple times throughout the proof, and together with Dickson's Lemma it establishes the existence of a sequence of configurations with growing counts for all states in Γ. This argument breaks in our case as Dickson's Lemma applies only if the number of states is constant. We overcome this problem by carefully choosing the thresholds on the counts that certain states reach.

Given a population protocol $\mathcal{P}_k$, a configuration $c : \Lambda_k \to \mathbb{N}$, and a function $g : \mathbb{N} \to \mathbb{N}^+$, we define sets $\Gamma_g(c) = \{s \in \Lambda_k \mid c(s) > g(|c|)\}$ and $\Delta_g(c) = \{s \in \Lambda_k \mid c(s) \leq g(|c|)\}$. Intuitively, $\Gamma_g(c)$ contains states above a certain count, while $\Delta_g(c)$ contains state below that count. Notice that $\Gamma_g(c) = \Lambda_k - \Delta_g(c)$.

The proof strategy is to first show that if a protocol stabilizes "too fast," then it can also reach configurations where all agents are in states in $\Gamma_g(c)$. Recall that a configuration c is defined as a function $\Lambda_k \to \mathbb{N}$. Let $S \subseteq \Lambda_k$ be some subset of states such that all agents in configuration c are in states from S, formally, $\{s \in \Lambda_k \mid c(s) > 0\} \subseteq S$. For notational convenience, we will write $c_{>0} \subseteq S$ to mean the same.

Theorem 2.7.10. *Consider a weakly monotonic population protocol $\mathcal{P}$. In case of leader election, define I to be a set of all input configurations. For majority, let I be a set of all input configurations with a fixed $\epsilon < 0.98$* [6]*. Suppose the protocol uses $k(n) \leq 1/2 \log\log n$ states for input configurations in I with n agents. Let $g : \mathbb{N} \to \mathbb{N}^+$ be a function such that $g(n) \geq 2^{k(n)}$ for all n and $6^{k(n)} \cdot k(n)^2 \cdot g(n) = o(n^{0.99})$.*

Suppose $\mathcal{P}$ stabilizes in $o\left(\frac{n}{(6^{k(n)} \cdot k(n)^3 \cdot g(n))^2}\right)$ time from all $i \in I$ with $|i| = n$. Then, for infinitely many n' such that the protocol uses the same number of states for when the number of agents is between n' and $3n'$, (i.e. $k(n') = \ldots = k(3n')$), there exists an initial configuration of $2n'$ agents $i_{2n'} \in I$ and a stable output configuration y of n' agents, such that for any configuration u that satisfies the boundedness predicate $\mathcal{B}(n', y)$ below, it holds that $i_{2n'} + u \Longrightarrow z$ where $z_{>0} \subseteq \Gamma_g(y)$.

We say that a configuration u satisfies the boundedness predicate $\mathcal{B}(n', y)$ if 1) it contains between 0 and n' agents, 2) all agents in u are in states from $\Delta_g(y)$, i.e. $u_{>0} \subseteq \Delta_g(y)$, and 3) $y(s) + u(s) \leq g(n')$ for all states $s \in \Delta_g(y)$.

Proof. For simplicity, set $b(n) = (6^{k(n)} + 2^{k(n)}) \cdot g(n)$ and $\beta(n) = k(n)^2 \cdot b(n) + k(n) \cdot b(n)$. Then, we know the protocol stabilizes in $o\left(\frac{n}{(\beta(n) \cdot k(n))^2}\right)$ time. By Lemma 2.7.4, for all sufficiently large n' we can find configurations of n' agents $i_{n'}, x, y : \Lambda_{k'} \to \mathbb{N}$, where $k' = k(n')$, such that

1. $i_{n'}$ is an initial configuration of the protocol $\mathcal{P}_{k'}$, (notice that the density requirement on $i_{n'}$ is satisfied for leader election and majority when $\epsilon < 0.98$).

2. $i_{n'} \Longrightarrow x \Longrightarrow_q y$, where y is a stable output configuration of n' agents for $\mathcal{P}_{k'}$ and q does not contain a β^2 bottleneck.

3. $\forall s \in \Lambda_{k'} : x(s) \geq \beta(n')$. Here, we use the assumption on function g.

Moreover, because the state complexity $k(n) \leq 1/2 \log\log n$ for sufficiently large n, and by weak monotonicity of the protocol, for infinitely many n' it additionally holds that the protocol uses k' states also for $n' + 1, n' + 2, \ldots, 3n'$ agents in the system. (otherwise state complexity would grow at least logarithmically in n).

[6] Interesting regime is when discrepancy ϵn is smaller than a constant fraction of the nodes.

Consider any such n'. Then, we can invoke Lemma 2.7.5 with x, y, transition sequence q and parameter $b = b(n')$. The definition of Δ in the lemma statement matches $\Delta_b(y)$, and β matches $\beta(n')$. Thus, we get an ordering of states $\Delta_b(y) = \{d_1, d_2, \dots, d_m\}$ and a corresponding sequence of transitions $\alpha_1, \alpha_2, \dots, \alpha_m$, where each α_j is of the form $(d_j, s_j) \to (o_j, o'_j)$ with $s_j, o_j, o'_j \notin \{d_1, d_2, \dots, d_j\}$. Finally, each transition α_j occurs at least $b(n') = (6^{k'} + 2^{k'}) \cdot g(n')$ times in q.

We will now perform transformations on the transition sequence q, called *surgeries*, with the goal of reaching a desired type of configuration. The next two claims specify these transformations, which are similar to the surgeries used in [DS15], but with some key differences due to configuration u and the new definitions of Γ and Δ. The proofs are provided later. Configuration u is defined as in the theorem statement. For brevity, we use $\Gamma_g = \Gamma_g(y)$, $\Delta_g = \Delta_g(y)$, $\Gamma_b = \Gamma_b(y)$ and $\Delta_b = \Delta_b(y)$.

Claim 2.7.11. *There exist configurations $e : \Lambda_{k'} \to \mathbb{N}$ and z' with $z'_{>0} \subseteq \Gamma_g$, such that $e + u + x \Longrightarrow z'$. Moreover, we have an upper bound on the counts of states in e: $\forall s \in \Lambda_{k'} : e(s) \le 2^{k'} \cdot g(n')$.*

The configuration $e + u + x$ has at most $2^{k'} \cdot g(n') \cdot k' + 2n'$ agents, which is less than $3n'$ for sufficiently large n'. Thus, the protocol $\mathcal{P}_{k'}$ is used.

For any configuration $e : \Lambda_{k'} \to \mathbb{N}$, let e^{Δ} be its projection onto Δ, i.e. a configuration consisting of only the agents from e in states Δ. We can define e^{Γ} analogously. By definition, $e^{\Gamma} = e - e^{\Delta}$.

Claim 2.7.12. *Let e be any configuration satisfying $\forall s \in \Lambda_{k'} : e(s) \le 2^{k'} \cdot g(n')$. There exist configurations p and w, such that $p_{>0} \subseteq \Delta_b$, $w_{>0} \subseteq \Gamma_g$ and $p + x \Longrightarrow p + w + e^{\Delta_g}$. Moreover, for counts in p, we have that $\forall s \in \Lambda_{k'} : p(s) \le b(n')$ and for counts in w^{Γ_g}, we have $\forall s \in \Gamma_g : w(s) \ge 2^{k'} \cdot g(n')$.*

Let our initial configuration $i_{2n'} \in I$ be $i_{n'} + i_{n'}$, which is a valid initial configuration and for majority, ϵ is also the same. Trivially, $i_{2n'} = i_{n'} + i_{n'} \Longrightarrow x + x$. Let us apply Claim 2.7.12 with e as defined in Claim 2.7.11, but use one x instead of p. This is possible because $\forall s \in \Lambda_{k'} : x(s) \ge \beta(n') \ge b(n') \ge p(s)$. Hence, we get

$x + x \Longrightarrow x + w + e^{\Delta_g} = x + e + (w - e^{\Gamma_g})$. The configuration $w - e^{\Gamma_g}$ is well-defined because both w and e^{Γ_g} contain agents in states in Γ_g, with each count in w being larger or equal to the respective count in e^{Γ_g}, by the bounds from the claims.

Finally, by Claim 2.7.11, we have $u + x + e + (w - e^{\Gamma_g}) \Longrightarrow z' + (w - e^{\Gamma_g})$. We denote the resulting configuration (with all agents in states in Γ_g) by z, and have $i_{2n'} + u \Longrightarrow z$, as desired. □

Claim 2.7.11. *There exist configurations $e : \Lambda_{k'} \rightarrow \mathbb{N}$ and z' with $z'_{>0} \subseteq \Gamma_g$, such that $e + u + x \Longrightarrow z'$. Moreover, we have an upper bound on the counts of states in e: $\forall s \in \Lambda_{k'} : e(s) \leq 2^{k'} \cdot g(n')$.*

Proof. The proof is similar to [DS15], but we consider a subsequence of the ordered transitions $\Delta_b = \{d_1, \ldots, d_m\}$ obtained earlier by Lemma 2.7.5. Since $b(n') \geq g(n')$, we can represent $\Delta_g = \{d_{j_1}, \ldots, d_{j_l}\}$, with $j_1 \leq \ldots \leq j_l$. We iteratively add groups of transitions at the end of the transition sequence q, (q is the transition sequence from x to y), such that, after the first iteration, the resulting configuration does not contain any agent in d_{j_1}. Next, we add group of transitions and the resulting configuration will not contain any agent agent in d_{j_1} or d_{j_2}, and we repeat this l times. In the end, no agents will be in states from Δ_g.

The transition ordering lemma provides us with the transitions to add. Initially, there are at most $g(n')$ agents in state d_{j_1} in the system (because of the requirement in Theorem 2.7.10 on counts in $u + y$). So, in the first iteration, we add the same amount, which is at most $g(n')$, of transitions $d_{j_1}, s_{j_1} \rightarrow o_{j_1}, o'_{j_1}$, after which, as $s_{j_1}, o_{j_1}, o'_{j_1} \notin \{d_1, \ldots d_{j_1}\}$, the resulting configuration will not contain any agent in configuration d_{i_1}. If there are not enough agents in the system in state s_{j_1} already to add all these transitions, then we add the remaining agents in state in s_{j_1} to e. For the first iteration, we may need to add at most $g(n')$ agents.

For the second iteration, we add transitions of type $d_{j_2}, s_{j_2} \rightarrow o_{j_2}, o'_{j_2}$ to the resulting transition sequence. Therefore, the number of agents in d_{j_2} that we may need to consume is at most $3 \cdot g(n')$, $g(n')$ of them could have been there in $y + u$, and we may have added $2 \cdot g(n')$ in the previous iteration, if for instance both o_{j_1} and

o'_{j_1} were d_{j_2}. In the end, we may need to add $3 \cdot g(n')$ extra agents to e.

If we repeat these iterations for all remaining $r = 3, \ldots, l$, in the end we will end up in a configuration z that contains all agents in states in Γ_g as desired, because of the property of transition ordering lemma that $s_{j_r}, o_{j_r}, o'_{j_r} \notin \{d_1, \ldots, d_{j_r}\}$. For any r, the maximum total number of agents we may need to add to e at iteration r is $(2^r - 1) \cdot g(n')$. The worst case is when o_{j_1} and o'_{j_1} are both d_{j_2}, and o_{j_2}, o'_{j_2} are both d_{j_3}, etc.

It must hold that $l < k'$, because the final configuration contains n' agents in states in Γ_g and none in $\{d_{j_1}, \ldots, d_{j_l}\}$, so Γ_g cannot be empty. Therefore, the total number of agents added to e is $g(n') \cdot \sum_{r=1}^{l}(2^r - 1) < 2^{l+1} \cdot g(n') \leq 2^{k'} \cdot g(n')$. This completes the proof because $e(s)$ for any state s can be at most the number of agents in e, which is at most $2^{k'} \cdot g(n')$. □

Claim 2.7.12. *Let e be any configuration satisfying $\forall s \in \Lambda_{k'} : e(s) \leq 2^{k'} \cdot g(n')$. There exist configurations p and w, such that $p_{>0} \subseteq \Delta_b$, $w_{>0} \subseteq \Gamma_g$ and $p + x \Longrightarrow p + w + e^{\Delta_g}$. Moreover, for counts in p, we have that $\forall s \in \Lambda_{k'} : p(s) \leq b(n')$ and for counts in w^{Γ_g}, we have $\forall s \in \Gamma_g : w(s) \geq 2^{k'} \cdot g(n')$.*

Proof. As in the proof of Claim 2.7.11, we define a subsequence $(j_1 \leq j_l)$, $\Delta_g = \{d_{j_1}, \ldots, d_{j_l}\}$ of $\Delta_b = \{d_1, \ldots, d_m\}$ obtained using Lemma 2.7.5. We start by the transition sequence q from configuration x to y, and perform iterations for $r = 1, \ldots m$. At each iteration, we modify the transition sequence, possibly add some agents to configuration p, which we will define shortly, and consider the counts of all agents not in p in the resulting configuration. Configuration p acts as a buffer of agents in certain states that we can temporarily borrow. For example, if we need 5 agents in a certain state with count 0 to complete some iteration r, we will temporarily let the count to -5 (add 5 agents to p), and then we will fix the count of the state to its target value, which will also return the "borrowed" agents (so p will also appear in the resulting configuration). As in [DS15], this allows us let the counts of certain states temporarily drop below 0.

We will maintain the following invariants on the count of agents, excluding the agents in p, in the resulting configuration after iteration r:

1) The counts of all states (not in p) in $\Delta_g \cap \{d_1, \ldots, d_r\}$ match to the desired counts in e^{Δ_g}.

2) The counts of all states in $\{d_1, \ldots d_r\} - \Delta_g$ are at least $2^{k'} \cdot g(n')$.

3) The counts in any state diverged by at most $(3^r - 1) \cdot 2^{k'} \cdot g(n')$ from the respective counts in y.

These invariants guarantee that we get all the desired properties after the last iteration. Let us consider the final configuration after iteration m. Due to the first invariant, the set of all agents (not in p) in states Δ_g is exactly e^{Δ_g}. All the remaining agents (also excluding agents in p) are in w, and thus, by definition, the counts of states in Δ_g in configuration w will be zero, as desired. The counts of agents in states $\Delta_b - \Delta_g = \{d_1, \ldots d_m\} - \Delta_g$ that belong to w will be at least $2^{k'} \cdot g(n')$, due to the second invariant. Finally, the counts of agents in Γ_b that belong to w will also be at least $b(n') - 3^{k'} \cdot 2^{k'} \cdot g(n') \geq 2^{k'} \cdot g(n')$, due to the third invariant and the fact that the states in Γ_b had counts at least $b(n')$ in y. Finally, the third invariant also implies the upper bound on counts in p. The configuration p will only contain the agents in states Δ_b, because the agents in Γ_b have large enough starting counts in y borrowing is never necessary.

In iteration r, we fix the count of state d_r. Let us first consider the case when d_r belongs to Δ_g. Then, the target count is the count of the state d_r in e^{Δ_g}, which we are given is at most $2^{k'} \cdot g(n')$. Combined with the third invariant, the maximum amount of fixing required may be is $3^{r-1} \cdot 2^{k'} \cdot g(n')$. If we have to reduce the number of d_r, then we add new transitions $d_r, s_r \rightarrow o_r, o'_r$, similar to Claim 2.7.11 (as discussed above, not worrying about the count of s_r possibly turning negative). However, in the current case, we may want to increase the count of d_r. In this case, we remove instances of transition $d_r, s_r \rightarrow o_r, o'_r$ from the transition sequence. The transition ordering lemma tells us that there are at least $b(n')$ of these transitions to start with, so by the third invariant, we will always have enough transitions to remove. We matched the count of d_r to the count in e^{Δ_g}, so the first invariant still holds. The second invariant holds as we assumed $d_r \in \Delta_g$ and since by Lemma 2.7.5, $s_r, o_r, o'_r \notin \{d_1, \ldots, d_r\}$. The third invariant also holds, because we performed at most $3^{r-1} \cdot 2^{k'} \cdot g(n')$ transition additions or removals, each affecting the count of any other given state by at most 2, and hence the total count differ by at most

$$(3^{r-1} - 1) \cdot 2^{k'} \cdot g(n') + 2 \cdot 3^{r-1} \cdot 2^{k'} \cdot g(n') = (3^r - 1) \cdot 2^{k'} \cdot g(n').$$

Now assume that d_r belongs to $\Delta_b - \Delta_g$. If the count of d_r is already larger than $2^{k'} \cdot g(n')$, than we do nothing and move to the next iteration, and all the invariants will hold. If the count is smaller than $2^{k'} \cdot g(n')$, then we set the target count to $2^{k'} \cdot g(n')$ and add or remove transitions as in the previous case, and the first two invariants will hold after the iteration. The only case when the count might require fixing by more than $(3^{r-1} - 1) \cdot 2^{k'} \cdot g(n')$ is when it originally was between $g(n')$ and $2^{k'} \cdot g(n')$ and decreased. Then, as in the previous case, the maximum amount of fixing required is at most $3^{r-1} \cdot 2^{k'} \cdot g(n')$ and considering the maximum effect on counts, the new differences can be at most $(3^r - 1) \cdot 2^{k'} \cdot g(n')$. As before, we also have enough transitions to remove and the third invariant holds. □

The following lemma is the key to getting a lower bound for a non-constant number of states.

Lemma 2.7.13. *Consider a population protocol $\mathcal{P}_k$ in a system of n agents, and an arbitrary fixed function $h : \mathbb{N} \to \mathbb{N}^+$ such that $h(n) \geq 2^k$. Let $\xi(n) = 2^k$. For all configurations $c, c' : \Lambda_k \to \mathbb{N}$, such that $c_{>0} \subseteq \Gamma_h(c) \subseteq \Gamma_\xi(c')$, any state producible from c is also producible from c'. Formally, for any state $s \in \Lambda_k$, $c \Longrightarrow y$ with $y(s) > 0$ implies $c' \Longrightarrow y'$ with $y'(s) > 0$.*

Proof. Since $h(n) \geq 2^k$, for any state from $\Gamma_h(c)$, its count in c is at least 2^k. As $\Gamma_h(c) \subseteq \Gamma_\xi(c')$, the count of each of these states in c' is also at least $\xi(n) = 2^k$. We say two agents have the same type if they are in the same state in c. We will prove by induction that any state that can be produced by some transition sequence from c, can also be produced by a transition sequence in which at most 2^k agents of the same type participate (ever interact). Configuration c only has agents with types (states) in $\Gamma_h(c)$, and configuration c' also has at least 2^k agents for each of those types, the same transition sequence can be performed from c' to produce the same state as from c, proving the desired statement.

The induction is as follows. There is a $\ell \leq k$, such that we can find sets $S_0 \subset S_1 \subset \ldots \subset S_\ell$ where S_ℓ contains all the states that are producible from c, and all sets S_j satisfy the following property. Let A_j be a set consisting of 2^j agents of each type in $\Gamma_h(c)$, out of all the agents in configuration c (we could also use c'), for the total of $2^j \cdot |\Gamma_h(c)|$ agents. There are enough agents of these types in c (and in c') as $j \leq \ell \leq k$. Then, for each $0 \leq j \leq \ell$ and each state $s \in S_j$, there exists a transition sequence from c in which only the agents in A_j ever interact and in the resulting configuration, one of these agents from A_j ends up in state s.

We do induction on j and for the base case $j = 0$ we take $S_0 = \Gamma_h(c)$. The set A_0 as defined contains one (2^0) agent of each type in $\Gamma_h(c) = S_0$ [7]. All states in S_0 are immediately producible by agents in A_0 via an empty transition sequence (without any interactions).

Let us now assume inductive hypothesis for some $j \geq 0$. If S_j contains all the producible states from configuration c, then $\ell = j$ and we are done. We will have $\ell \leq k$, because $S_0 \neq \varnothing$ and $S_0 \subset S_1 \subset \ldots S_\ell$ imply that S_ℓ contains at least ℓ different states, and there are k total. Otherwise, there must be some state $s \notin S_j$ that can be produced after an interaction between two agents both in states in S_j, let us say by a transition $\alpha : r_1, r_2 \to s, p$ with $r_1, r_2 \in S_j$ (or there is no state that cannot already be produced). Also, as S_j contains at least j states out of k total, and there is the state $s \notin S_j$, $j < k$ holds and the set A_{j+1} is well-defined. Let us partition A_{j+1} into two disjoint sets B_1 and B_2 where each contain 2^j agents from c for each type. Then, by induction hypothesis, there exists a transition sequence where only the agents in B_1 ever interact and in the end, one of the agents $b_1 \in B_1$ ends up in

[7] In c, all the agents are in one of the states of $\Gamma_h(c)$, so as long as $n > 0$ there must be at least one agent per state (type). So, if $\Gamma_h(c) = \varnothing$, then n must necessarily be 0, so nothing is producible $A_0 = \varnothing$, $\ell = 0$ and we are done.

the state r_1. Analogously, there is a transition sequence for agents in B_2, after which an agent $b_2 \in B_2$ ends up in state r_2. Combining these two transition and adding one instance of transition α in the end between agents b_1 and b_2 (in states r_1 and r_2 respectively) leads to a configuration where one of the agents from A_{j+1} is in state s. Also, all the transitions are between agents in A_{j+1}. Hence, setting $S_{j+1} = S_j \cup \{s\}$ completes the inductive step and the proof. □

We are now ready to prove the lower bounds for majority and leader election as separate corollaries of the main theorem.

Corollary 2.7.14. *Any weakly monotonic population protocol that stably computes correct majority decision with state complexity* $s(n, \epsilon) \leq 1/2 \log \log n$, *must take* $\Omega\left(\frac{n}{36^{s(n,\epsilon)} \cdot s(n,\epsilon)^6 \cdot \max(2^{s(n,\epsilon)}, \epsilon n)^2}\right)$ *expected parallel time to stabilize.*

Proof. We set $g(n) = \max(2^{s(n,\epsilon)+1}, 10\epsilon n)$. For majority computation, initial configurations consist of agents in one of two states, with the majority state holding an ϵn advantage in the counts. The bound is nontrivial only in a regime $\epsilon n \in o(\sqrt{n})$, which we will henceforth assume without loss of generality.

Assume, for contradiction, that some protocol stabilizes in expected parallel time $o\left(\frac{n}{36^{s(n,\epsilon)} \cdot s(n,\epsilon)^6 \cdot \max(2^{s(n,\epsilon)}, \epsilon n)^2}\right)$. Since $\epsilon n \in o(\sqrt{n})$, we have $\epsilon < 0.98$. Using Theorem 2.7.10 we can find infinitely many configurations i and z of at most $3n'$ agents, such that (1) $i + u \Longrightarrow z$, (2) i is an initial configuration of $2n'$ agents with majority state A and advantage $2\epsilon n'$. (3) The number of states used for $n', n'+1, \ldots, 3n'$ is the same, denoted by k'. (4) $z_{>0} \subseteq \Gamma_g(y)$, i.e all agents in z are in states that have counts at least $g(n')$ in some stable output configuration y of n' agents.

Suppose initial majority is A. Let us prove that for all sufficiently large n', in any stable configuration y of n' agents, strictly less than $2^{s(n',\epsilon)} \leq g(n')/2$ agents must be in the initial minority state B. The reason is that if c is the initial configuration of all n' agents in state B by weak monotonicity, the protocol must stabilize from c to decision Win_B. By Lemma 2.7.13, from any configuration that contains at least $2^{s(n',\epsilon)}$ agents in B it would also be possible to reach a configuration where some agent supports decision B. Therefore, all stable final configuration y have at most

$g(n')/2-1$ agents in initial minority state B. This allows us to let u be a configuration of $g(n')/2+1 \geq 5\epsilon n'+1$ agents in state B.

To get the desired contradiction we will prove two things. First, z is actually a stable output configuration for decision Win_A (majority opinion in i), and second, $i+u$ is an initial configuration from which the protocol $\mathcal{P}_{k'}$ must stabilize to the correct majority decision. However, $i+u$ has more nodes in state B. This will imply that the protocol stabilize to a wrong outcome, and complete the proof by contradiction.

If we could reach a configuration from z with any agent in a state s that maps to output Win_B ($\gamma(s) = Win_B$), then by Lemma 2.7.13, from a configuration y (which contains $2^{s(m,\epsilon)}$ agents in each of the states in $\Gamma_g(y)$) we can also reach a configuration with an agent in a state s that maps to output Win_B. However, configuration y is a final stable configuration majority decision A.

Configuration i contains $2\epsilon n'$ more agents in state A states than in state B. Configuration u consists of at least $5\epsilon n'+1$ agents all in state B. Hence, $i+u$, which is a valid initial configuration (as all nodes are in states A or B) of at most $3n'$ agents, has a majority of state B with discrepancy at least $3\epsilon n'$. By weak monotonicity, the protocol $\mathcal{P}_{k'}$ must stabilize to the decision WIN_B from $i+u$, as desired. □

Corollary 2.7.15. *Any weakly monotonic population protocol that stably elects at least one and at most $\ell(n)$ leaders with state complexity $s(n) \leq 1/2 \log\log n$, must take $\Omega\left(\frac{n}{144^{s(n)} \cdot s(n)^6 \cdot \ell(n)^2}\right)$ expected parallel time to stabilize.*

Proof. We set $g(n) = 2^{s(n)} \cdot \ell(n)$. The set I of initial configurations for leader election all agents are in the same starting state.

Suppose for contradiction that a protocol stabilizes in expected parallel time $o\left(\frac{n}{144^{s(n)} \cdot s(n)^6 \cdot \ell(n)^2}\right)$. Using Theorem 2.7.10 and setting u to be a configuration of zero agents, we can find infinitely many configurations i and z of $2n'$ agents, such that (1) $i \Longrightarrow z$, (2) $i \in I$ is an initial configuration of $2n'$ agents, (3) The number of states used for $n', n'+1, \ldots, 2n'$ is the same, denoted by k'. (4) $z_{>0} \subseteq \Gamma_g(y)$, i.e. all agents in z are in states that each have counts of at least $2^{s(n')} \cdot \ell(n')$ in some stable output configuration y of n' agents.

Because y is a stable output configuration of a protocol that elects at most $\ell(n')$ leaders, none of these states in $\Gamma_g(y)$ that are present in strictly larger counts ($2^{s(n')} \cdot \ell(n') > \ell(n')$) in y and z can be leader states (i.e. γ maps these states to output *Lose*). Therefore, the configuration z does not contain a leader. This is not sufficient for a contradiction, because a leader election protocol may well pass through a leaderless configuration before stabilizing to a configuration with at most $\ell(n')$ leaders. We prove below that any configuration reachable from z must also have zero leaders. This implies an infinite time on stabilization from a valid initial configuration i (as $i \Longrightarrow z$) and completes the proof by contradiction.

If we could reach a configuration from z with an agent in a leader state, then by Lemma 2.7.13, from a configuration c' that consists of $2^{s(n')}$ agents in each of the states in $\Gamma_g(y)$, it is also possible to reach a configuration with a leader, let us say through transition sequence q. Recall that the configuration y contains at least $2^{s(n')} \cdot \ell(n')$ agents in each of these states in $\Gamma_g(y)$, hence there exist disjoint configurations $c'_1 \subseteq y$, $c'_2 \subseteq y$, etc, $\ldots, c'_{\ell(n')} \subseteq y$ contained in y and corresponding transition sequences $q_1, q_2, \ldots, q_{\ell(n')}$, such that q_j only affects agents in c'_j and leads one of the agents in c'_j to become a leader. Configuration y is a output configuration so it contains at least one leader agent already, which does not belong to any c'_j because as mentioned above, all agents in c'_j are in non-leader states. Therefore, it is possible to reach a configuration from y with $\ell(n') + 1$ leaders via a transition sequence q_1 on the c'_1 component of y, followed by q_2 on c'_2, etc, $q_{\ell(n')}$ on $c'_{\ell(n')}$, contradicting that y is a stable output configuration. □

```
Parameters:
$\rho$, an integer $> 0$, set to $\Theta(\log n)$
$T_c < \rho$, an integer $> 0$, threshold for clock-creation
State Space:
  $WorkerStates =$        $.phase \in \{1, \ldots, 2\log n + 1\}$,
                          $.value \in \{0, 1/2, 1\}$
                          $.preference \in \{WIN_A, WIN_B\}$;
  $ClockStates =$         $.position \in \{0, \Psi - 1\}$,
                          $.preference \in \{WIN_A, WIN_B\}$;
  $BackupStates =$        4 states from the protocol of [DV12];
  $TerminatorStates =$    $\{D_A, D_B\}$.
Additional two bit-flags in every state   $.InitialState \in \{A, B\}$
                                          $.clock\text{-}creation \in \{true, false\}$;
Input: States of two nodes, $S_1$ and $S_2$
Output: Updated states $S_1' = \mathsf{update}(S_1, S_2)$ and $S_2' = \mathsf{update}(S_2, S_1)$
Auxiliary Procedures:
$backup(S) = \begin{cases} A_{[DV12]} & \text{if } S.InitialState = A; \\ B_{[DV12]} & \text{otherwise.} \end{cases}$
$term\text{-}preference(S) = \begin{cases} D_A & \text{if } S = D_A \textbf{ or } S.preference = WIN_A \\ D_B & \text{if } S = D_B \textbf{ or } S.preference = WIN_B \end{cases}$
$pref\text{-}conflict(S, O) = \begin{cases} true & term\text{-}preference(S) \neq term\text{-}preference(O) \\ false & \text{otherwise.} \end{cases}$
$is\text{-}strong(S) = \begin{cases} true & \text{if } S \in WorkerStates \textbf{ and } S.value \neq 0 \\ false & \text{otherwise.} \end{cases}$
$clock\text{-}label(O) = \begin{cases} 0 & \text{if } O.position \in [2\rho, 3\rho) \\ 1 & \text{if } O.position \in [0, \rho) \\ -1 & \text{otherwise.} \end{cases}$
$inc\text{-}phase(\phi, O) = \begin{cases} true & \text{if } \phi = O.phase - 1 \textbf{ or } \phi \bmod 2 = 1 - clock\text{-}label(O) \\ false & \text{otherwise.} \end{cases}$
8  procedure update$\langle S, O \rangle$
9      if $S \in BackupStates$ or $O \in BackupStates$ then
10         if $S \in BackupStates$ and $O \in BackupStates$ then
11             $S' \leftarrow \mathsf{update}_{[DV12]}(S, O)$
12         else if $O \in BackupStates$ then
13             $S' \leftarrow backup(S)$
14         else $S' \leftarrow S$
15         return $S'$
       // Backup states processed, below $S$ and $O$ are not in backup states
16     if $S \in TerminatorStates$ or $O \in TerminatorStates$ then
17         if $pref\text{-}conflict(S, O) = false$ then
18             $S' \leftarrow term\text{-}preference(S)$
19         else $S' \leftarrow backup(S)$
20         return $S'$
```

Figure 2-2: Pseudocode for the phased majority algorithm, part 1/2

```
   // Below, both $S$ and $O$ are workers or clocks
21 $S' \leftarrow S$
22 if $O.\textit{clock-creation} = \textit{false}$ then
23     $S'.\textit{clock-creation} \leftarrow \textit{false}$
24 if $\textit{is-strong}(S) = \textit{false}$ and $\textit{is-strong}(O) = \textit{true}$ then
25     $S'.\textit{preference} \leftarrow O.\textit{preference}$
   // Clock creation flag and preference updated (always)
26 if $S \in \textit{ClockStates}$ then
27     if $O \in \textit{ClockStates}$ then
           /* Update $S'.\textit{Position}$ according to Section 2.3.  If gap between
              $S.\textit{position}$ and $O.\textit{position}$ not less than $\rho$, set $S' \leftarrow \textit{backup}(S)$.
              If $S.\textit{position} \geq T_c$, set $S'.\textit{clock-creation} \leftarrow \textit{false}$.   */
28     return $S'$
   // Below, $S$ is a worker and $O$ is a worker or a clock
29 $\phi \leftarrow S.\textit{phase}$
30 if $\textit{inc-phase}(\phi, O) = \textit{true}$ then
31     if $\phi = 2\log n + 1$  or  $(\phi \bmod 2 = 0$  and  $S.\textit{value} = 1)$ then
32         $S' \leftarrow \textit{term-preference}(S)$
33     else
34         $S.\textit{phase} = \phi + 1$
35         if $\phi \bmod 2 = 0$  and  $S.\textit{value} = 1/2$ then
36             $S'.\textit{value} = 1$
37     return $S'$
38 if $O \in \textit{ClockStates}$ then
39     return $S'$
   // Below, $S$ is a worker and $O$ is a worker
40 if $|S.\textit{phase} - O.\textit{phase}| > 1$ then
41     $S' \leftarrow \textit{backup}(S)$
42     return $S'$
   // Below, worker meets worker within the same phase
43 if $\phi \bmod 2 = 1$ then
       // Cancellation phase
44     if $S.\textit{value} = 1$  and  $O.\textit{value} = 1$  and  $\textit{pref-conflict}(S, O) = \textit{true}$ then
45         if $S'.\textit{clock-creation} = \textit{true}$  and  $S.\textit{preference} = \textit{WIN}_A$ then
46             $S' \leftarrow \textit{clock}(.\textit{position} = 0, .\textit{preference} = S.\textit{preference})$
47         else
48             $S'.\textit{value} \leftarrow 0$
49 else
       // Doubling phase
50     if $S.\textit{value} + O.\textit{value} = 1$ then
51         $S'.\textit{value} = 1/2$
52 return $S'$
```

Figure 2-3: Pseudocode for the phased majority algorithm, part 2/2

Chapter 3

Shared Memory

3.1 Anonymous Space Lower Bound

The optimal space complexity of consensus in asynchronous shared memory was an open problem for two decades. For a system of n processes, no algorithm using a sublinear number of registers is known. Up until very recently, the best known lower bound due to Fich, Herlihy, and Shavit was $\Omega(\sqrt{n})$ registers.

Fich, Herlihy, and Shavit first proved their lower bound for the special case of the problem where processes are anonymous (i.e. they run the same algorithm) and then extended it to the general case.

Here we close the gap for the anonymous case of the problem. We show that any consensus algorithm from read-write registers for anonymous processes that satisfies the standard *nondeterministic solo termination* property, has to use $\Omega(n)$ registers in some execution. This implies an $\Omega(n)$ lower bound on the space complexity of deterministic obstruction-free and randomized wait-free consensus, matching the upper bound. As in [FHS98, AE14b], the bound is for the worst-case space complexity of the algorithm, i.e. for the number of registers used in some execution, regardless of its actual probability.

Our argument relies heavily on the anonimity of the processes. We introduce new techniques for marshalling anonymous processes and their executions, in particular, the concepts of *leader-follower pairs* and *reserving executions*, that play a critical role in the lower bound argument and will hopefully be more generally applicable.

3.1.1 Definitions and Notation

We consider the standard asynchronous shared-memory model with anonymous processes and atomic read-write registers. Processes take steps, where each step is either a shared-memory step or an internal step. There are no guarantees on when a process takes its next step, in fact, any process is allowed stop taking steps altogether.

A shared-memory step of a process is either a read or a write to some register. With an internal step, a process can make local nondeterministic choices, or return a value, after which the process takes no more steps. Naturally, the outcomes of nondeterministic choices influence the state of the process and its next steps.

A *configuration* is a collection of all process states and the contents of all registers, and describes the global state of the system. An *execution* is a sequence of steps by processes and a *solo execution* is an execution where all steps are taken by a single process. An execution starts in a configuration and leads to a configuration. We will use capital latin letters to denote configurations and lower case greek letters to denote executions, and we will refer to the configuration reached by an execution α that started in configuration C as the configuration $C\alpha$. Finally, an execution $\alpha\beta$ simply stands for the execution α followed by the execution β.

Notice that, if a process p in state s makes a certain nondeterministic choice and ends up in state s', then, at any time, any process q that is in the same state s might make the same nondeterministic choice and also end up in state s'. In this work, we will restrict our attention to executions where all processes in the same state make the same nondeterministic choices.

We will also only consider executions where any process, after a shared-memory step, immediately performs all subsequent internal steps, leading to a shared-memory step, unless it returns a value. Therefore, from now on, the term *step* will refer exclusively to a shared-memory step, and we will consider only process states in which the next step is a shared-memory step.

In a system of anonymous processes, all processes with the same input start in the same state. If in some configuration, a process p in state s writes v to some register r and changes state to s', then in any configuration, any process q in the same state s will also write v to register r with its next step and change state to s'. If in some configuration, a process p in state s reads v from register r and changes state to s', then in any configuration, any process q in state s will also read from the register r with its next step. Notice that the reads by p and q are in different configurations and might return different results. However, if q happens to read the value v, then it will also change its state to s'. The above statements are true since by our assumption p and q make the same nondeterministic choices.

We say that a process p is *covering* a register r, if the next step of p is a write to r. A *block write* of a set of processes P to a set of covered registers R is a sequence of write steps by processes in P, where each step writes to a different register in R and all registers in R get written to.

A *clone* of a process p, exactly as in [FHS98, AE14b], is defined as another process with the same input as p, that shadows p by performing the same steps as p in lockstep, reading and writing the same values immediately after p, and remaining in the same state, until just before some write of p. Because the system consists of anonymous processes, in any execution with sufficiently many processes, for any write step of p, there always exists an alternative execution with a clone q that shadowed p all the way until the write. In particular, in the alternative execution, process q *covers* the register and is about to write the value that p last wrote there. Moreover, the two executions with or without the clone covering the register are completely indistinguishable to all processes other than the clone itself.

In the binary consensus problem, each participating process starts with a binary input, 0 or 1, and is supposed to return a binary output. The correctness requirement is that all outputs must be the same and equal to the input of some process. We say that an execution *decides* 0 (or 1) if some process returns 0 (or 1, respectively) during this execution.

The wait-free termination requirement means that each participating process is supposed to eventually return an output within a finite number of its own steps, regardless of how the other processes are scheduled. The obstruction-free termination requirement means that any process that runs solo is supposed to eventually return an output within a finite number of steps. In randomized algorithms, processes are allowed to flip random coins and decide their next steps accordingly. In this setting, the randomized wait-free termination requirement means that each participating process is supposed to eventually return an output within a finite number of its own steps with probability 1.

The FLP result shows that, in the asynchronous shared memory model with read-write registers, no deterministic algorithm can solve binary consensus in a wait-free way. However, it is possible to solve consensus both in a deterministic obstruction-free and in a randomized wait-free way. The *nondeterministic solo termination* property means that from each reachable configuration, for each process, there exists a finite solo execution by the process where it terminates and returns an output. We prove our lower bounds for binary consensus algorithms that satisfy the nondeterministic solo termination property, because both deterministic obstruction-free algorithms and randomized wait-free algorithms fall into this category.

The following standard *indistinguishability lemma* is a key ingredient in our lower bound arguments.

Lemma 3.1.1. *If a process p has the same state in two configurations C and D, and the contents of all registers are the same in both of these configurations, then every solo execution by p starting from C can have the same results starting from D.*

3.1.2 A Square-Root Lower Bound

To demonstrate our approach, we start by presenting a proof of the $\Omega(\sqrt{n})$ space lower bound in the anonymous setting. This is the same as the best known lower bound from [FHS98], but the inductive argument and the valency definition used are considerably different.

If there is a solo execution of some process returning 0 from a configuration, then we call this configuration *0-solo-deciding* (and *1-solo-deciding* if there is a solo execution of a process that returns 1). Solo termination implies that every configuration is 0-solo-deciding or 1-solo-deciding. Note that the same configuration can be simultaneously 0-solo-deciding and 1-solo-deciding. We call such configurations *solo-bivalent*, and *solo-univalent* otherwise. If a configuration is 0-solo-deciding, but not 1-solo-deciding (i.e. no solo execution from this configuration decides 1), then we call it *0-solo-valent*, meaning that the configuration is solo-univalent with valency 0. Analogously, a configuration is *1-solo-valent* if it is 1-solo-deciding but not 0-solo-deciding.

Lemma 3.1.2. *Consider a system of at least two processes. Then, in every solo-bivalent configuration, we can always find two distinct processes p and q, such that there is a solo execution of p returning 0 and a solo execution of q returning 1.*

Proof. Either the configuration is solo-bivalent because of solo executions of distinct processes, in which case we are done, or two solo executions of some process p return different values. In this case it suffices to consider any terminating solo execution of another process q. □

Lemma 3.1.3. *Consider a system of $\frac{(r-1)r}{2} + 2$ anonymous processes for any $r \geq 0$. Then, for any consensus algorithm that uses atomic read-write registers and satisfies the nondeterministic solo termination property, there exists a configuration C_r reachable by an execution ρ_r with the following properties:*

- *There is a set R of r registers, each of which has been written to during ρ_r, and*
- *the configuration C_r is solo-bivalent.*

Proof. The proof is by induction, with the base case $r = 0$. Our system consists of two processes p and q, p starts with input 0, q starts with input 1, and C_0 is the initial configuration. The solo-bivalency of C_0 follows from Lemma 3.1.1 as process p cannot distinguish between C_0 and a configuration where both processes start with input 0, in which case the terminating solo execution of p (which exists due to the nondeterministic solo termination property) is required to return 0. Analogously, some solo execution of process q returns 1.

Let us assume the induction hypothesis for some r and prove it for $r+1$. We can reach a solo-bivalent configuration C_r using $\frac{(r-1)r}{2}+2$ processes by an execution ρ_r that writes to a set R of r registers. The goal is to use another set of r processes in order to write to a new register and extend C_r to C_{r+1}.

From configuration C_r, by Lemma 3.2.12, there exists a solo execution α by process p that returns 0 and a solo execution β of process q that returns 1. For each register in R, let us add a new clone of the process that last wrote to it. Hence, each register in R will now be covered by a clone, poised to write the same value as present in the register in configuration C_r.

Let us now apply a covering argument utilizing the clones. Consider execution $\rho_r\alpha\gamma\beta$, where γ is a block write to R by the new clones. We know that process p returns 0 after $\rho_r\alpha$. During its solo execution α, process p has to write to a register outside of R. Otherwise, the configuration after $\rho_r\alpha\gamma$ is indistinguishable from C_r to process q as the values in all registers are the same, and q is still in the same state as in C_r. Hence, by Lemma 3.1.1, q can return 1 after $\rho_r\alpha\gamma\beta$ as it would after $\rho_r\beta$, contradicting the correctness of the consensus algorithm. Analogously, process q has to write outside of R during β. Let $\alpha=\alpha' w_p \alpha''$, where w_p is the first write of p outside the set of registers R, and let $\beta=\beta' w_q \beta''$, with w_q being the first write outside of R. Let ℓ be the length of $\gamma\beta' w_q$ and, for $0 \le i \le \ell$, let π_i be the length i prefix of $\gamma\beta' w_q$.

Next, we use a valency argument to reach C_{r+1}. We show that either the configuration reached after $\rho_r\alpha'\gamma\beta' w_q$, or one of the configurations reached after $\rho_r\alpha'\pi_i w_p$ for some i, satisfies the properties necessary to be C_{r+1}. The number of processes used in any case is $\frac{(r-1)r}{2}+2$ from before, plus r clones introduced during the inductive step, which gives $\frac{r(r+1)}{2}+2$ as required. Moreover, in any of these configurations we can find $r+1$ registers that have been written to during ρ_{r+1}, i.e. while reaching C_{r+1} from the initial configuration, as we can include r registers in R and one more register written to by either w_p or w_q. Thus, we only need to show that one of these configurations is solo-bivalent.

Assume the contrary. Then the configuration after $\rho_r\alpha'\pi_0 w_p$ is solo-valent. Moreover, since C_r is the configuration reached by ρ_r, π_0 is the empty execution, and p returns 0 in $\rho_r \alpha' w_p \alpha''$, the configuration $C_r \alpha' \pi_0 w_p$ is actually 0-solo-valent. On the other hand, the configuration reached by $\rho_r \alpha' \gamma \beta' w_q = \rho_r \alpha' \pi_\ell$ must be 1-solo-valent. It is solo-univalent by the contradiction assumption and 1-solo-deciding as q cannot distinguish configurations C_r and $C_r \alpha' \gamma$, and thus by Lemma 3.1.1, the solo execution β'' of q can still return 1.

Because the configuration $C_r \alpha' \pi_\ell$ is 1-solo-valent, any terminating solo execution of process p from that configuration must also return 1. In particular, every terminating solo execution that starts by p performing its next step w_p returns 1. So the configuration $C_r \alpha' \pi_\ell w_p$ must be 1-solo-valent: a terminating solo execution of p returns 1 and it is solo-univalent by the contradiction assumption for $i = \ell$. Therefore, configuration $C_r \alpha' \pi_i w_p$ is 0-solo-valent for $i = 0$ and 1-solo-valent for $i = \ell$. Hence, there exists an i, such that $X = C_r \alpha' \pi_i w_p$ is 0-solo-valent, and $Y = C_r \alpha' \pi_{i+1} w_p$ is 1-solo-valent. Let o be the extra step in π_{i+1}.

o is either performed by process q or by the new clones as a part of block write γ. It may not be a read or a write to the same register as w_p writes to, since p would not distinguish between X and Y and by Lemma 3.1.1, could return the same output from both configurations. This would contradict the different solo-univalencies of X and Y. Therefore, steps w_p and o commute. Let σ be a terminating solo execution from Y by the process f that performed o. Since Y is 1-solo-valent, f returns 1 after σ. Now consider f performing its next step o from X. Since w_p and o commute, the process cannot distinguish the resulting configuration Xo from Y and by Lemma 3.1.1, it returns 1 after solo execution σ from configuration Xo. However, $o\sigma$ is also a solo execution by f from X that returns 1, contradicting the 0-valency of X. The contradiction proves the induction step, completing our argument. □

Notice that for n processes, Lemma 3.1.3 directly implies the existence of an execution where $\Omega(\sqrt{n})$ registers are written to, proving the desired lower bound.

3.1.3 Linear Lower Bound

Consider a system with n anonymous processes and an arbitrary correct consensus algorithm satisfying the nondeterministic solo termination property. We will assume that no execution of the algorithm uses $\lfloor n/14-1 \rfloor$ registers and derive a contradiction. For notational convenience, let us define m to be $\lfloor n/14 \rfloor - 2$.

The argument in Lemma 3.1.3 in the previous section relies on a new set of clones in each iteration to overwrite the changes to the contents of the registers made during the inductive step. This is the primary reason why we only get an $\Omega(\sqrt{n})$ lower bound. As the authors of [FHS98] also mention, to get a stronger lower bound we would instead have to reuse processes to overwrite the registers. However, after the overwriting, we cannot guarantee that processes would still cover various registers. Moreover, it is insufficient to simply cover registers with processes without any knowledge of what they are about to write. We start by introducing concepts that will be used to overcome these challenges.

Process Pairs: The Leader and The Follower

To prove Lemma 3.1.3, we used clones that covered registers to overwrite these registers using a block-write, to reset their contents to be the same as in a previous configuration with known valency. In order to do something similar, but without using new clones, we will consider pairs of processes, consisting of a leader process and a follower process. The follower is a clone of the leader process and the pair remain in the same states and perform the same steps during the whole execution. Every process in the system will be either a leader or a follower in some pair.

Usually, when we schedule a leader to perform a step, its follower performs the same step immediately after the leader. In this case, we say that *the pair perfomed the step.* However, sometimes we will *split* the pair by having only the leader perform a write step and let the follower cover the register. We will explicitly say when this is the case. After we split the pair in such a way, we will delay scheduling the leader and the follower will remain poised to write to the covered register. Later, we will schedule the follower to write, effectively resetting the register to the value it had after the write by the leader. As the leader did not take any steps in the meanwhile, after the write the follower will again be in the same state as its leader. Hence, the pair of the leader and the follower will no longer be split, and will continue taking steps in lock-step as a pair.

This is very different from the way clones were used in the proof of Lemma 3.1.3, because after the pair of the leader and its follower is united, it can be split again. Therefore, the same follower can reset the contents of registers written by its leader multiple times.

We call a split pair of a leader and a follower *fresh* as long as the register that the leader wrote to, and its follower is covering, has not been overwritten. After the register is overwritten, we call the split pair *stale*. In any configuration, there is at most one fresh split pair whose follower covers a particular register.

In addition, we also use cloning in a way similar to the proof of Lemma 3.1.3, except that we do this only a constant number of times, as opposed to r times, to reach the next configuration C_{r+1}. Moreover, each time we do this, we actually clone a pair, i.e. we create duplicates of both a leader and its follower. The new leader-follower pair is in the same state as the original pair, and from there on behaves as any other pair.

By definition, both the leader and the follower in unsplit pairs are in the same state. Therefore, we say that an unsplit pair covers a register when both the leader and the follower cover it and we say that an unsplit pair returns when both the leader and the follower return in two successive steps. A solo execution by an unsplit pair p is an execution containing even number of steps, where the leader in pair p takes a step, immediately followed by exactly the same step of the follower in p. Thus, nondeterministic solo termination property for the leader process implies nondeterminstic solo termination for the executions of the unsplit pair.

Reserving Executions

Intuitively, *reserving executions* ensure that, for each register written to during an execution, some pair is reserved to cover it. We can use these pairs for covering in subsequent inductive configurations.

Definition 3.1.4. *Let C be some configuration reachable by the algorithm, and let P be a set of at least $m + 1$ unsplit pairs. We call an execution γ that starts from configuration C* reserving *from C by P if:*

- *γ is a sequence of steps by pairs in P (first by the leader of the pair, immediately followed by the follower). Hence, γ contains even number of steps.*
- *For each prefix γ' of γ and for each register written to during γ', in configuration $C\gamma'$, there is an unsplit pair $p \in P$ that covers it.*
- *If a pair $p \in P$ returns during γ, then this happens during the last two steps of γ.*

Notice that, by definition, any even-length prefix of a reserving execution is also a reserving execution. Let $\mathsf{Res}(C, P)$ be the set of all reserving executions from C by pairs in P that end with a pair $p \in P$ returning. We first show that, given sufficiently many pairs, such an execution exists. This will be essential for defining valency based on reserving executions. Recall that we assumed a strict upper bound of m on the number of registers that can ever be written.

Lemma 3.1.5. *For any reachable configuration C and any set P of at least $m + 1$ unsplit pairs that have not returned, we have that $\mathsf{Res}(C, P) \neq \varnothing$.*

Proof. For a given C and P, we will prove the lemma by constructing a particular reserving execution γ that ends when some pair $p \in P$ returns. We start with an empty γ and repeatedly extend it.

In the first stage, consider each pair $p \in P$, one at a time. By nondeterministic solo termination, there exists a solo execution by pair p where p returns. If this solo execution contains write steps, extend γ by the prefix of the execution before the first write, and consider the next pair in P. Otherwise, complete γ by extending it with the whole solo execution of p.

The first stage consists of extending the execution at most $|P|$ times. Each time, we extend γ by a prefix of a finite solo execution of some pair $p \in P$. These steps are reads by leaders and followers of pairs in P, and therefore the constructed prefix of γ is reserving.

If some pair returns in the first stage, the construction of γ is complete. Otherwise, since the first stage is finite, we move on to the second stage.

In the second stage, the execution γ is extended by repeatedly doing the following two phases. At the beginning of the first phase, we keep an invariant that each of the at least $m + 1$ pairs in P is covering a register (this holds after the first stage). Let R be the set of registers covered by pairs in P. Since $|R| \leq m < |P|$, we can find two pairs $p, q \in P$ covering the same register in R. By nondeterministic solo termination, there exists a solo execution of pair p where p returns. If this solo execution contains a write to a register outside of R, extend γ by the prefix of the execution before this write, and continue from the first phase. Note that p still covers a register, satisfying the invariant. Add this register to R. Otherwise, complete γ by extending it with the whole solo execution of pair p.

In the second stage, each iteration extends γ by a finite number of steps. After each iteration, if the construction is not complete, the size of R increases by one, but it cannot become more than m as only m registers can ever be written. Thus, after at most m finite extensions, we will complete the construction of γ when some pair returns.

To see that the execution is reserving, notice that all registers that were written to are in R. By construction, each register in R remains covered from the beginning of the second stage or when it is first added to R during the second stage. $\square$

The next lemma follows immediately from the definition of reserving executions.

Lemma 3.1.6. *Consider a reachable configuration C, a set of at least $m + 1$ unsplit pairs P' none of which have returned in C, and some unsplit pair $p \notin P'$ poised to perform a write w_p in C. Let C' be the configuration reached from C after the pair*

p performs w_p (first the leader, then the follower). Moreover, assume that another unsplit pair $q \neq p$ with $q \notin P'$ is covering the same register that w_p writes to. If $\gamma \in \mathsf{Res}(C', P')$, then $w_p w_p \gamma$ is in $\mathsf{Res}(C, P' \cup \{p\} \cup \{q\})$.

New Definition of Valency

Here we define valency based on reserving executions. For a set of process pairs U, we say that a configuration C is *0-reserving-deciding$_U$*, if there exists a subset of $m+1$ unsplit pairs $P \subseteq U$ ($|P| = m+1$), and a reserving execution in $\mathsf{Res}(C, P)$ returning 0. We define *1-reserving-deciding$_U$* analogously. Using Lemma 3.1.5 it immediately follows that

Lemma 3.1.7. *Let U contain at least $m+1$ unsplit pairs that have not returned in configuration C. Then, C is v-reserving-deciding$_U$ for at least one $v \in \{0, 1\}$.*

A configuration that is both 0-reserving-deciding$_U$ and 1-reserving-deciding$_U$ is called *reserving-bivalent$_U$*. Otherwise, the configuration is called *reserving-univalent$_U$*.

If a configuration is 0-reserving-deciding$_U$, but not 1-reserving-deciding$_U$ (i.e. no reserving execution by $m+1$ unsplit pairs in U starting from this configuration decides 1), then we call it *0-reserving-valent$_U$*. Analogously, a configuration is called *1-reserving-valent$_U$* if it is 1-reserving-deciding$_U$, but not 0-reserving-deciding$_U$.

The next lemma says that, from a bivalent configuration, there are reserving executions by disjoint sets of processes that decide different values.

Lemma 3.1.8. *Let C be a reserving-bivalent$_U$ configuration for U a set of at least $3m+2$ unsplit pairs. Then there are disjoint sets of $m+1$ unsplit pairs $P' \subseteq U$ and $Q' \subseteq U$ ($P' \cap Q' = \varnothing$), such that an execution in $\mathsf{Res}(C, P')$ returns 0 and an execution in $\mathsf{Res}(C, Q')$ returns 1.*

Proof. None of the pairs in U have already returned in configuration C, as that would contradict the existence of a reserving execution returning the other output. As C is reserving-bivalent$_U$, there exist sets of $m+1$ unsplit pairs P and Q, such that an execution in $\mathsf{Res}(C, P)$ returns 0 and an execution in $\mathsf{Res}(C, Q)$ returns 1. If P and

Q do not intersect then we are done by setting $P' = P$ and $Q' = Q$.

Otherwise, consider an arbitrary set $H \subseteq U - P - Q$ of $m+1$ unsplit pairs. If C is 0-reserving-deciding$_H$, we set $P' = H$ and $Q' = Q$, and if C is 1-reserving-deciding$_H$, then we set $P' = P$ and $Q' = H$. One of these cases holds due to Lemma 3.1.7, completing the proof. $\square$

The Main Proof

Consider any correct consensus algorithm satisfying nondeterministic solo termination in a system of anonymous processes, with the property that every execution uses at most m registers. We will restrict attention to executions in which the processes are partitioned into leader-follower pairs. Let $0 \leq r \leq m$. Suppose there exists a set U containing $5m + 6 + 2r$ leader-follower pairs and a configuration C_r that is reachable by an execution by leaders and followers in U.

Consider configuration C_r. Let R_s denote the set of registers that are covered by the follower of a fresh split pair. Suppose that there are no stale split pairs. Suppose there is a set R_c of $r - |R_s|$ other registers that are each covered by at least one unsplit pair. Let V consist of r leader-follower pairs covering the r registers in $R_c \cup R_s$. In particular, all split pairs are in V. Finally, suppose that there are two disjoint sets of unsplit pairs $P, Q \subseteq U - V$, such that some execution $\alpha \in \mathsf{Res}(C_r, P)$ returns 0, some execution $\beta \in \mathsf{Res}(C_r, Q)$ returns 1, and $|P| + |Q| \leq 2m + 4$. Recall that by definition of reserving executions, in both α and β, all steps are taken as pairs - first the leader, then its follower, and the last two steps are some leader-follower returning the output.

Let C_0 be an initial configuration that contains a set U of $5m + 6$ leader-follower pairs, half of which have input 0 and half of which have input 1. Let R_s, R_c, and V be empty. Let P be a set of $m + 1$ pairs in U with input 0 and let Q be a set of $m + 1$ pairs in U with input 1. There are no split pairs in any initial configuration. By Lemma 3.1.5, $\mathsf{Res}(C_0, P), \mathsf{Res}(C_0, Q) \neq \varnothing$. Since all steps of executions in $\mathsf{Res}(C_0, P)$ are by processes with input 0, all executions in $\mathsf{Res}(C_0, P)$ return 0. Similarly, all executions in $\mathsf{Res}(C_0, Q)$ return 1. Thus, it is possible to satisfy all the assumptions when $r = 0$.

We will construct a set U' of $5m+6+2(r+1)$ leader-follower pairs, a configuration C_{r+1} that is reachable by an execution by leaders and followers in U', disjoint sets of registers R'_s and R'_c and disjoint sets of leader-follower pairs $V', P', Q' \subseteq U'$ such that, in C_{r+1},

1. R'_s is the set of registers that are covered by the follower of a fresh split pair in V',
2. one unsplit pair in V' covers each register in R_c,
3. $|V'| = |R'_s| + |R'_c| = r + 1$,
4. some execution in $\mathsf{Res}(C_{r+1}, P')$ returns 0,
5. some execution in $\mathsf{Res}(C_{r+1}, Q')$ returns 1,
6. $|P'| + |Q'| \leq 2m + 4$, and
7. there are no stale split pairs.

We will construct the execution that starts in C_r and reaches C_{r+1}. In $U' - U$ we have two more pairs available that have not taken steps and can be used to clone a leader-follower pair. Let T denote $U - V - P - Q$. Since $|V| = r$ and $|P| + |Q| \leq 2m + 4$, we have $|T| \geq 3m + 2 + r \geq 3m + 2$.

In C_r, each register in R_c is covered by some pair (both a leader and its follower) in V. Let γ_c be a block write to all registers in R_c by only the leaders but not the followers of the respective covering pairs, i.e. after each write we get a new fresh split pair. Without loss of generality assume that $C_r\gamma_c$ is 1-reserving-deciding$_T$.

For any execution δ by processes not in V, we denote by $W(\delta)$ the set of registers written to during δ. In C_r, each register in R_s is covered by a follower of a split pair in V whose leader has already performed the write and is stopped. For any execution δ' in which processes in V do not take steps, we define $\gamma_s(\delta')$ to be the block write to all registers in $R_s \cap W(\delta')$ (registers written to during δ' that are in R_s) by the respective followers of split pairs in V. After each write, another follower catches up with its leader and a previously split pair is united. So, if we run an execution δ' from C_r that changes the contents of some registers in R_s, we can clean these changes up by executing $\gamma_s(\delta')$, which leads to all registers in R_s having the same contents as in

C_r.

Using a crude covering argument we can show that

Lemma 3.1.9. *The execution α must contain a write step outside $R_c \cup R_s$.*

Proof. Assume the contrary. We know that the execution α starting from C_r returns 0. Only processes in P take steps during α, P is disjoint from V and only processes in V take steps during γ_c and $\gamma_s(\alpha)$. T is disjoint from both P and V, and thus the configurations $C_r\alpha\gamma_s(\alpha)\gamma_c$ and $C_r\gamma_c$ are indistinguishable to all processes in T. This is because no process in T has taken steps, the registers in $R_c \cup R_s$ contain the same values, and no other registers have been written to during α, $\gamma_s(\alpha)$ or γ_c. Configuration $C_r\gamma_c$ is 1-reserving-deciding$_T$, hence the same execution from $\mathsf{Res}(C_r\gamma_c, T)$ that returns 1, also returns 1 when executed from $C_r\alpha\gamma_s(\alpha)\gamma_c$. This contradicts the correctness of the consensus algorithm. □

Let us write $\alpha = \alpha' w_p w_p \alpha''$, where w_p is the first write step to a register $\mathsf{reg} \notin (R_c \cup R_s)$, performed by some leader-follower pair $p \in P$. Next, we prove the following technical lemma using a FLP-like case analysis:

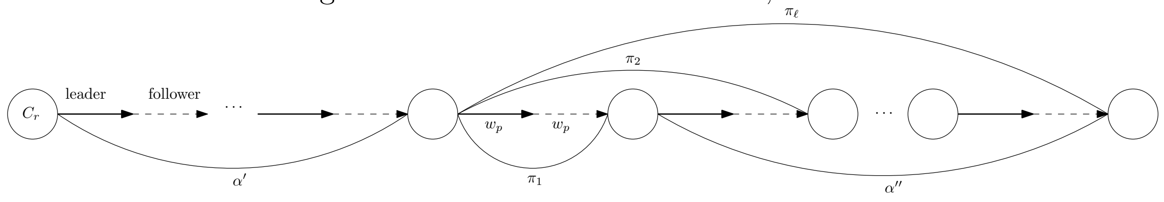
Lemma 3.1.10. *We can construct an execution ρ, such that in configuration $C_r\rho$, we have disjoint sets of registers R'_s, R'_c and disjoint sets of pairs $V', P', Q' \in U'$, that satisfy the first six properties. By these properties all fresh split pairs in $C_r\rho$ belong to V', but some pairs in $U' - V' - P' - Q'$ might be split and stale, contradicting the seventh property. However, for each such pair p*

- *$p \in V$, it was split on a register $x \in R_s$ and was fresh in C_r, and*
- *neither the leader nor follower in p have taken steps in ρ, but ρ includes a write to x that makes p stale.*

Proof. Recall that T does not contain any split pairs.

Case 1: the configuration $C_r\alpha'$ is 1-reserving-deciding$_T$: Let ℓ be the length of $w_p w_p \alpha''$ and let π_j be a prefix of $w_p w_p \alpha''$ of length $2j$ for $0 \le j \le \ell/2$. Hence, the difference between π_j and π_{j+1} is the same step performed twice by a leader and a

Figure 3-1: Proof of Lemma 3.1.10, Case 1



follower of a pair $p \in P$, as illustrated in Figure 3-1. Pairs in P are not split, as by the inductive hypothesis only pairs in V are split, and $V \cap P = \varnothing$. We consider two further subcases.

Case 1.1: for some $0 \leq j \leq \ell/2$, the configuration $C_r\alpha'\pi_j$ is reserving-bivalent$_T$: In this case, we let ρ be $\alpha'\pi_j$. Since $C_r\rho$ is reserving-bivalent$_T$ and T contains at least $3m+2$ unsplit pairs, by Lemma 3.1.8, there are $P' \subseteq T$ and $Q' \subseteq T$, with $P' \cap Q' = \varnothing$ and $|P'| = |Q'| = m+1$, such that an execution in $\mathsf{Res}(C_r\rho, P')$ returns 0 and an execution in $\mathsf{Res}(C_r\rho, Q')$ returns 1.

Case 1.2: for every $0 \leq i \leq \ell/2$, the configuration $C_r\alpha'\pi_i$ is reserving-univalent$_T$: By ***Case 1*** assumption $C_r\alpha'\pi_0$ is 1-reserving-deciding$_T$, hence, it must be 1-reserving-valent$_T$. On the other hand, α returns 0, so the configuration $C_r\alpha'\pi_{\ell/2}$ must be 0-reserving-valent$_T$. No intermediate configuration is bivalent, so we can find a configuration $C_r\alpha'\pi_j$ that is 1-reserving-valent$_T$, while $C_r\alpha'\pi_{j+1}$ is 0-reserving-valent$_T$, for some $0 \leq j < \ell/2$. We let ρ be $\alpha'\pi_j$.

Let oo be the steps by a leader-follower pair in P separating π_j and π_{j+1}. o may not be a read, as no process in T could distinguish between a 1-reserving-valent$_T$ configuration $C_r\alpha'\pi_j$ and a 0-reserving-valent$_T$ configuration $C_r\alpha'\pi_j oo = C_r\alpha'\pi_{j+1}$.

Let Q' be a set of any $m+1$ pairs in T. By Lemma 3.1.5, $\mathsf{Res}(C_r\rho, Q')$ is non-empty and since $C_r\rho$ is 1-reserving-valent$_T$, all executions in $\mathsf{Res}(C_r\rho, Q')$ return 1. Recall that U' contains the pairs in U and an additional two leader-follower pairs that have not taken any steps. Let us use these pairs p' and p'' to clone the leader-follower pair performing the write step o. Both cloned leaders and followers will be in the same state as the leader and follower in the original pair performing o. The leader and follower in p' are thus poised to perform write steps o' identical to the step o at configuration $C_r\rho$, and the leader and the follower in p'' are poised to perform o'' identical to o.

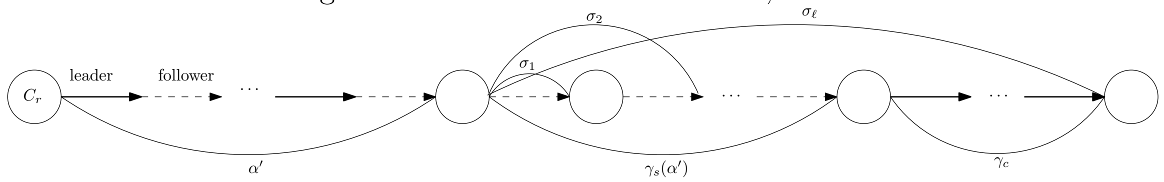
Let F be a set of $m+1$ pairs from $T - Q'$ (we know $|T| \geq 3m+2$). Let P' be $F \cup \{p', p''\}$, $|P'| = m+3$ in total. By Lemma 3.1.5 and the fact that configuration $C_r\alpha\pi_{j+1}$ is 0-reserving-valent$_T$, there is a reserving execution $\xi \in \mathsf{Res}(C_r\alpha\pi_{j+1}, F)$ that returns 0.

Having pair p' perform o' from $C_r\rho$ while p'' covers the same register with the step o'', we reach the configuration $C_r\rho o' o'$. This configuration is indistinguishable from $C_r\alpha\pi_{j+1}$ for any pair in F, because they have not taken steps and the contents of all registers are the same. Thus, execution ξ from $C_r\rho o' o'$ also returns 0, i.e. $o' o' \xi$ from $C_r\rho$ returns 0, and by Lemma 3.1.6, $o' o' \xi \in \mathsf{Res}(C_r\alpha\pi_j, P') = \mathsf{Res}(C_r\rho, P')$. By construction $|P'| + |Q'| = 2m+4$ and $P' \cap Q' = \varnothing$. The reason why we cloned two pairs instead of cloning one pair and using a pair from P that was poised to perform o is that we later require $P', Q' \subseteq T \cup (U' - U)$, in order to ensure that P' and Q' are disjoint from V', constructed below.

Case 1: the rest of the proof for both subcases The set R'_s will be $R_s - W(\rho)$, i.e. the registers from R_s that have not been written to during the execution $\rho = \alpha'\pi_j$ from C_r. For each of these registers we still have the same pair from V split on it, and since this pair was fresh in C_r and the register has not been written to during ρ, it is still fresh in $C_r\rho$ as required. There are no other fresh split pairs in $C_r\rho$: no new split pairs were introduced during ρ, and the rest of fresh pairs in C_r were split on $R_s \cap W(\rho)$. These pairs are no longer fresh in $C_r\rho$, as the registers their followers covered were written to during ρ.

The set R'_c is simply $(R_c \cup R_s \cup \{\mathsf{reg}\}) - R'_s = R_c \cup (R_s \cap W(\rho)) \cup \{\mathsf{reg}\}$. We must show that there is a leader-follower pair covering each of these registers in configuration $C_r\rho$. For each register in R_c, we take the same pair from V that was covering it in C_r. For each register in $(R_s \cap W(\rho)) \cup \{\mathsf{reg}\}$, we find a pair from P

Figure 3-2: Proof of Lemma 3.1.10, Case 2



covering it in $C_r\rho$. Since α is a reserving execution from C_r, all its prefixes of even length including $\alpha'\pi_j$ are also reserving. Thus, in $C_r\alpha'\pi_j = C_r\rho$, for each register that has been written to during ρ, in particular for registers in $(R_s \cap W(\rho)) \cup \{\mathsf{reg}\}$, we find a covering pair in P. Technically, if $j = 0$, register reg is not yet written, but the next step in α is w_p by a pair covering reg.

The set $V' \subseteq V \cup P$ contains all $r+1$ pairs we used to cover registers in $R_c' \cup R_s' = \mathsf{R}_c \cup R_s \cup \mathsf{reg}$. Recall that by our construction, $P', Q' \subseteq T \cup (U' - U)$. Also, T was disjoint from V, P and Q, and so are the two pairs in $U' - U$ that had never taken steps before. Therefore, we have $(P' \cup Q') \cap V' = \varnothing$ as required.

Case 2: the configuration $C_r\alpha'$ is 0-reserving-valent$_T$: No processes in V take steps in α', and therefore $\gamma_s(\alpha')$ is well defined. In $\gamma_s(\alpha')$, each register in $R_s \cap W(\alpha')$ is overwritten to the value that the register had in C_r by the follower of the split pair in V that covered it. Hence, for all processes in T, configuration $C_r\alpha'\gamma_s(\alpha')\gamma_c$ is indistinguishable from the 1-reserving-deciding$_T$ configuration $C_r\gamma_c$. This is because the processes in T have not taken steps (T is disjoint from $P \cup V$) and the contents of all registers are the same in these configurations (α' contains writes only to registers in $R_c \cup R_s$).

Let us denote by ℓ the length of execution $\gamma_s(\alpha')\gamma_c$ and let σ_j be the prefix of this execution of length j for $0 \leq j \leq \ell$. Notice that, unlike the previous case, where the difference between π_j and π_{j+1} was a pair of identical steps by a leader and the follower, the difference between σ_j and σ_{j+1} is exactly one step, by either a leader or a follower of some pair. By definition, each step in $\gamma_s(\alpha')$ is performed by a follower (uniting a previously split pair after each step), while each step in γ_c

is performed by a leader (creating a new fresh split pair after each step). This is illustrated in Figure 3-3.

Case 2.1: for some $0 \leq j \leq \ell$, the configuration $C_r\alpha'\sigma_j$ is reserving-bivalent$_T$: We let ρ be $\alpha'\sigma_j$. As in ***Case 1.1***, $C_r\sigma_j$ is reserving-bivalent$_T$ and $|T| \geq 3m+2$. Therefore, by Lemma 3.1.8, there are $P' \subseteq T$ and $Q' \subseteq T$, with $P' \cap Q' = \varnothing$ and $|P'| = |Q'| = m+1$, such that an execution in $\mathsf{Res}(C_r\rho, P')$ returns 0 and an execution in $\mathsf{Res}(C_r\rho, Q')$ returns 1.

Case 2.2: for every $0 \leq i \leq \ell$, the configuration $C_r\alpha'\sigma_i$ is reserving-univalent$_T$: The configuration $C_r\alpha'\sigma_0$ is 0-reserving-valent$_T$ by the ***Case 2*** assumption. Since all writes in α' are to registers in $R_c \cup R_s$, the contents of all registers are the same in $C_r\alpha'\sigma_\ell$ and $C_r\gamma_c$. Moreover, the processes in T have not taken steps in $\alpha'\sigma_\ell$ or γ_c. Therefore, configuration $C_r\alpha'\sigma_\ell$ is indistinguishable from 1-reserving-deciding$_T$ $C_r\gamma_c$ to all processes in T.

No intermediate configuration is bivalent, so there is a configuration $C_r\alpha'\sigma_j$ that is 0-reserving-valent$_T$, while $C_r\alpha'\sigma_{j+1}$ is 1-reserving-valent$_T$, for some $0 \leq j < \ell$. We let ρ be $\alpha'\sigma_j$.

We can construct P' and Q' in a very similar way to ***Case 1.2***. If o is the step separating σ_j and σ_{j+1}, o may not be a read as before, and we use two pairs in $U' - U$ to clone leader-follower pairs p' and p'', both about to perform identical write steps o' and o''. We let P' be a set of any $m+1$ pairs in T and Q' be a set of $m+3$ pairs, with $m+1$ pairs from $T - P'$ and the two cloned pairs. Then, by the same argument as in ***Case 1.2***, P' and Q' satisfy all required properties.

Case 2: the rest of the proof for both subcases The set R'_s is $(R_s - W(\alpha')) \cup (R_c \cap W(\sigma_j))$. It consists of registers from R_s that were not written to during α' and registers from R_c that were written to during σ_j (during the prefix of block write γ_c that was executed).

For each register in $R_s - W(\alpha')$, we still have the same pair from V split on it in configuration $C_r\rho$ as in C_r. This split pair was fresh in C_r and covered a register in R_s that has not been written to during $\rho = \alpha'\sigma_j$. Therefore, it is still fresh as required in $C_r\rho$. Execution σ_j may contain a prefix of γ_c, during which only leaders but not the followers take steps and new fresh split pairs are created. These pairs are split on registers in $R_c \cap W(\sigma_j)$, and there is one newly split fresh pair per each of these registers. No other split pairs are fresh, since fresh pairs that were split on $R_s \cap W(\alpha')$ cannot be fresh in $C_r\rho$, as the registers covered by their followers were written to during α'.

The set R'_c is $(R_c - W(\sigma_j)) \cup (R_s \cap W(\alpha')) \cup \{\mathsf{reg}\}$. As in ***Case 1***, α' is a prefix of a reserving execution $\alpha \in \mathsf{Res}(C_r, P)$, so for each register in $R_s \cap W(\alpha')$ there is a covering pair from P in $C_r\rho$. The register reg is covered by the leader-follower pair in P with a pending write w_p. For each register in $R_c - W(B_j)$, we take the pair from V that was covering it in C_r. Neither leader nor follower in this pair have taken steps during $\rho = \alpha'\sigma_j$ and still cover the same register in $C_r\rho$.

As in ***Case 1***, the set $V' \subseteq V \cup P$ contains all $r+1$ pairs used to cover registers in $R'_c \cup R'_s = R_c \cup \mathsf{R}_s \cup \mathsf{reg}$. Also, $P', Q' \subseteq T \cup (U' - U)$ and it follows as before that $(P' \cup Q') \cap (V') = \varnothing$ as required. □

In order to finish the proof, we need to show how to get rid of stale split pairs that might exist in configuration $C_r\rho$. In C_r there were no stale split pairs in the system, so they must have appeared during the execution ρ constructed in Lemma 3.1.10 For each register in $R_s \cap W(\rho)$, there is a split pair $p \in V$ that was fresh in C_r, but is stale in $C_r\rho$, because the register covered by the follower of p was overwritten during ρ.

We now modify the execution ρ; For each split pair p that is stale in $C_r\rho$, let s be the first write step during ρ to the register the follower of p covers. We add a write step by the follower of p in execution ρ immediately before step s. This way, no pair other than the follower observes a difference, since the changes are promptly overwritten by s. The follower, meanwhile, catches up with the leader, hence, the pair is united.

Let ρ' be the modified execution, where as described above, we have added a write step of a follower of each pair that was stale in $C_r\rho$ (and covered a register in $R_s \cap$

$W(\rho))$. We will use the configuration $C_r\rho'$ as C_{r+1}. Because of the indistinguishability, C_{r+1} satisfies all the required properties and does not contain any stale split pairs, as all such pairs from $C_r\rho$ are now united.

Corollary 3.1.11. *In a system of n anonymous processes, any consensus algorithm satisfying non-deterministic solo termination must use $\lfloor n/14 \rfloor - 1$ registers.*

Proof. Suppose for the sake of contradiction that all executions use at most $m = \lfloor n/14 \rfloor - 2$ registers, where n is the number of anonymous processes. Then, we can reach C_{m+1} using $10m + 12 + 4m + 4 < n$ processes. In configuration C_{m+1}, there are $m + 1$ registers in $R_c \cup R_s$, each of which has either already been written to (R_s) or is covered by an unsplit process (R_c). □

3.2 The Space Hierarchy

3.2.1 Model

Our model is similar to the standard asynchronous shared memory model [AW04], albeit with a few important differences. We consider a system of $n \geq 2$ processes that *supports* some set of deterministic synchronization *instructions*, $\mathcal{I}$, on a set of identical memory locations. The processes take steps at arbitrary, possibly changing, speeds and may crash at any time. Each step is an atomic invocation of some instruction on some memory location. Scheduling is controlled by an adversary.

The processes can use instructions on the memory locations to *simulate* (or *implement*) various objects. An object provides a set of operations which processes can call. Although a memory location together with the supported instructions can be viewed as an object, we do not do so, to emphasize the *uniformity requirement* that the same set of instructions is supported on all memory locations.

We consider the problem of solving *obstruction-free m-valued consensus* in such a system. Here, each of the n processes has an input from $\{0, 1, \ldots, m-1\}$ and is supposed to output a value (called a *decision*), such that all decisions are the same and equal to the input of one of the processes. Obstruction-free means that every process will eventually decide a value provided no other process is taking steps at the same time. When $m = n$, we call this problem *n-consensus.*

For lower bounds, we consider n processes solving 2-valued consensus, which is also called binary consensus. In deterministic algorithms, the next step of a process is uniquely defined at every configuration, and process p is said to be *poised* to perform a specific instruction on a memory location r at a configuration C if p's next step in configuration C is to perform that instruction on r. We prove our lower bounds for a more general, *nondeterministic solo termination* [FHS98] property, which means that from each reachable configuration, for each process, there exists a finite solo execution by the process where it terminates and returns an output. In these lower bounds, for each process p in a configuration C, we always consider one fixed next step δ of p, such that p has a finite terminating solo execution from $C\delta$. Then, we say that p is *poised* on a memory location r at a configuration C if δ is an instruction on r.

3.2.2 Arithmetic Instructions

Consider a system that supports only *read*() and either *add*(x), *multiply*(x), or *set-bit*(x). We show how to solve n-consensus using a single memory location in such a system. The idea is to show that we can simulate certain collections of objects that can solve n-consensus.

An *m-component unbounded counter* object has m components, each with a non-negative integral value. It supports an *increment*() operation on each component, that increments the value of the component by 1, and a *snapshot*() operation, that returns the values of all m components. In the next lemma, we present a *racing counters* algorithm that bears some similarity to a consensus algorithm by Aspnes and Herlihy [AH90].

Lemma 3.2.1. *It is possible to solve obstruction-free m-valued consensus among n processes using an m-component unbounded counter.*

Proof. We initialize all components with 0 and associate a separate component c_v with each possible input value v. Processes that *promote* a particular value v increment

c_v. (Initially, each process promotes its input value.) After performing an increment, a process takes a snapshot of all m components and promotes the value associated with a highest component (breaking ties arbitrarily). When a process observes that some component c_v is more than n larger than all other components, it returns the value v associated with c_v. This works because each other process will increment some component at most once before next taking a snapshot, and, in that snapshot, c_v will still be the only maximum. From then on, each of these processes will promote value v and keep incrementing c_v, which will soon become large enough for all processes to return v. Obstruction-freedom follows because a process running on its own will continue to increment the same component and, thus, eventually be able to return. □

The component values may grow arbitrarily large in the preceding protocol. The next lemma shows that it is possible to overcome this limitation when each component also supports a *decrement*() operation. More formally, an *m-component bounded counter* object has m components, where each component has a value in $\{0, 1, \ldots, 3n-1\}$. It supports both *increment*() and *decrement*() operations on each component, along with a *snapshot*() operation that returns the values of all components. If a process ever attempts to increment a component that has value $3n - 1$ or decrement a component that has value 0, the object breaks (and every subsequent operation invocation returns $\perp$).

Lemma 3.2.2. *It is possible to solve obstruction-free m-valued consensus among n processes using an m-component bounded counter.*

Proof. We modify the construction in Lemma 3.2.1 slightly by changing what a process does when it wants to increment c_v to promote the value v. Among the other components (i.e. excluding c_v), let c_u be one that is highest. If $c_u < n$, it increments c_v, as before. If $c_u \geq n$, then, instead of incrementing c_v, it decrements c_u.

A component with value 0 is never decremented. This is because, after the last time some process observed it to be at least n, each process will decrement the component at most once before reading its value again. Similarly, a component c_v never becomes larger than $3n - 1$: after the last time some process observed it to be less than $2n$, each process can increment c_v at most once before reading its value again. If $c_v \geq 2n$, then either the other components are less than n, in which case the process returns without incrementing c_v, or the process decrements some other component, instead of incrementing c_v. $\square$

In the following theorem, we show how to simulate unbounded and bounded counter objects.

Theorem 3.2.3. *It is possible to solve n-consensus using a single memory location that supports only read() and either multiply(x), add(x), or set-bit(x).*

Proof. We first give an obstruction-free implementation using a single location in a system with *read*() and *multiply*(x) instructions. This proves the claim for this set of instructions, by Lemma 3.2.1. The location is initialized with 1. For each component c_v, where $v \in \{0, \ldots, m-1\}$, let p_v be the $(v+1)$'st prime number. A process increments a component c_v by performing *multiply*(p_v). A *read*() instruction returns the value x currently stored in the memory location, which gives a snapshot of all components: component c_v is the exponent of p_v in the prime decomposition of x.

A similar construction does not work in a system with *read*() and *add*(x) instructions. For example, suppose one component is incremented by calling *add*(a) and another component is incremented by calling *add*(b). Then, the value ab can be obtained by incrementing the first component b times or incrementing the second component a times. However, we can use a single memory location that supports $\{$*read*(), *add*(x)$\}$ to implement an m-component bounded counter. By Lemma 3.2.2, this is sufficient for solving consensus. We initialize the location with 0 and view the value stored in the location as a number written in base $3n$. We interpret the i'th least significant digit of this number as the value of the component c_{i-1}. To increment c_i, we perform *add*($(3n)^i$), to decrement c_i, we perform *add*($-(3n)^i$), and *read*() provides an atomic snapshot of all components.

Finally, in systems supporting *read*() and *set-bit*(x), we partition the memory location into blocks of mn bits. Process i sets the $(vn+i)$'th bit in block b to indicate that it has incremented component c_v at least b times. To increment component c_v, a process sets its bit for component c_v in the appropriate block based on the number of times it has previously incremented c_v. It is possible to determine to current value of each component via a single *read*(): the value of component c_v is simply the sum of the number of times each process has incremented c_v. □

3.2.3 Increment

Consider a system that supports only *read*, *write*(x), and either *fetch-and-increment*() or *increment*(). We prove that it is not possible to solve n-consensus in the first (stronger) case using a single memory location and we provide an algorithm in the second (weaker) case, which uses $O(\log n)$ memory locations.

Theorem 3.2.4. *It is not possible to solve nondeterministic solo terminating consensus for $n \geq 2$ processes using a single memory location that supports only read(), write(x), and fetch-and-increment().*

Proof. Suppose there is a binary consensus algorithm for two processes, p and q, using only 1 memory location. Consider solo terminating executions α and β by p with input 0 and input 1, respectively. Let α' be the longest prefix of α that does not contain a *write* and define β' analogously. Without loss of generality, suppose that in β' at least as many *fetch-and-increment* instructions are performed as in α'. Let C be the configuration that results from executing α' starting from the initial configuration in which p has input 0 and the other process, q has input 1.

Consider the shortest prefix β'' of β' in which p performs the same number of fetch-and-increments as it performs in α'. Let C' be the configuration that results from executing β'' starting from the initial configuration in which both p and q have input 1. Then q must decide 1 in its solo terminating execution γ starting from configuration C'. However, C and C' are indistinguishable to process q, so it must decide 1 in γ starting from configuration C. Thus, p cannot have decided yet in configuration C, otherwise both 0 and 1 would have been decided.

Therefore, p is poised to perform a *write* in configuration C. Let α'' be the remainder of α, so $\alpha = \alpha'\alpha''$. Since there is only one memory location, the configurations resulting from performing this *write* starting from C and $C\gamma$ are indistinguishable to p. Thus, p also decides 0 starting from $C\gamma$. But in this execution, both 0 and 1 are decided. This is a contradiction. □

The following well-known construction converts any algorithm for solving binary consensus to an algorithm for solving n-valued consensus [HS12].

Lemma 3.2.5. *Consider a system that supports a set of instructions that includes read() and write(x). If it is possible solve obstruction-free binary consensus among n processes using only c memory locations, then it is possible to solve n-consensus using only $(c+2)\cdot\lceil\log_2 n\rceil - 2$ locations.*

Proof. The processes agree bit-by-bit in $\lceil\log_2 n\rceil$ asynchronous rounds, each using $c+2$ locations. A process starts in the first round with its input value as its value for round 1. In round i, if the i'th bit of its value is 0, a process writes this value in a designated 0-location for the round. Otherwise, it writes it in a designated 1-location. Then, it performs the obstruction-free binary consensus algorithm using c locations to agree on the i'th bit v of the output. If this bit differs from the i'th bit of its value, the process reads one of the recorded values from the designated v-location for round i and adopts its value for the next round. Note that some process must have already recorded a value to this location since, otherwise, the bit $\bar{v}$ would have been agreed upon. This ensures that the values used for round $i+1$ are all input values and they all agree in their first i bits. By the end, all processes have agreed on $\lceil\log_2 n\rceil$ bits, i.e. on one of the at most n different input values.

We can save two locations because the last round does not require designated 0 and 1-locations. □

We can implement a 2-component unbounded counter, defined in Section 3.2.2, using two locations that support *read*() and *increment*(). The values in the two locations never decrease. Therefore, a *snapshot*() operation of the counter can be performed using the double collect algorithm in [AAD+93]. By Lemma 3.2.1, a 2-component unbounded counter lets n processes solve obstruction-free binary consensus. The next result then follows from Lemma 3.2.5.

Theorem 3.2.6. *It is possible to solve n-consensus using only $O(\log n)$ memory locations that support only read(), write(x), and increment().*

3.2.4 Buffers

In this section, we consider the instructions *ℓ-buffer-read*() and *ℓ-buffer-write*(x), for $\ell \geq 1$, which generalize read and write, respectively. Specifically, an *ℓ-buffer-read*() instruction returns the last ℓ inputs to *ℓ-buffer-write* instructions previously applied on the memory location, in order from least recent to most recent.

We consider a system that supports the instruction set $\mathcal{B}_\ell = \{\ell\text{-}buffer\text{-}read(), \ell\text{-}buffer\text{-}write(x)\}$, for some $\ell \geq 1$. We call each memory location in such a system an *ℓ-buffer*. Note that a 1-buffer is simply a register. For $\ell > 1$, an ℓ-buffer essentially maintains a buffer of the ℓ most recent writes to that location and allows them to be read.

In Section 3.2.4, we show that a single ℓ-buffer can be used to simulate a powerful *history* object that can be updated by at most ℓ processes. This will allow us to simulate an obstruction-free variant of Aspnes and Herlihy's algorithm for n-consensus [AH90] and, hence, solve n-consensus, using only $\lceil n/\ell \rceil$ ℓ-buffers. In Section 3.2.4, we prove that $\lceil (n-1)/\ell \rceil$ ℓ-buffers are necessary, which matches the upper bound whenever $n-1$ is not a multiple of ℓ.

Simulations Using Buffers

A *history* object H supports two operations, *get-history*() and *append*(x), where *get-history*() returns the sequence of all values appended to H by prior *append* operations, in order. We first show that, using a single ℓ-buffer B, we can simulate a history object H that supports at most ℓ different appenders, but arbitrarily many readers.

Lemma 3.2.7. *A single ℓ-buffer can simulate a history object on which at most ℓ different processes can perform append(x) and any number of processes can perform get-history().*

Proof. Without loss of generality, assume that no value is appended to H more than once. This can be achieved by having a process include its process identifier and a sequence number along with the value that it wants to append.

In our implementation, B is initially $\perp$ and each value written to B is of the form $(\mathbf{h}, x)$, where $\mathbf{h}$ is a history of appended values and x is a single appended value.

To implement *append*(x) on H, a process obtains a history $\mathbf{h}$ by performing *get-history*() on H and then performs ℓ-*buffer-write* on B with value $(\mathbf{h}, x)$. The operation is linearized at this ℓ-*buffer-write* step.

To implement *get-history*() on H, a process simply performs an ℓ-*buffer-read* of B to obtain a vector $(a_1, \ldots, a_\ell)$, where a_ℓ is the most recently written value. The operation is linearized at this step.

Next, we consider a given *get-history*() operation. Let R be the ℓ-*buffer-read* step, which is the linearization point of this *get-history*() operation. We describe how the return value of this operation is computed, and prove that in all cases, it returns the sequence of inputs to all *append* operations on H linearized before R, in order from least recent to most recent. The proof assumes that all *get-history*() operations for which strictly less *append* operations on H were linearized before their respective linearization points, return the correct output (i.e. the proof is by induction on the number of *append* operations linearized before the *get-history*() operation). Let $(a_1, \ldots, a_\ell)$ be the vector returned by R.

We have $(a_1, \ldots, a_\ell) = (\perp, \ldots, \perp)$ only when there are no ℓ-*buffer-write* steps before R, i.e. if and only if no *append* operations are linearized before R. In this case, the empty sequence is returned, as required.

Now, suppose that there is at least one *append* operation linearized before R and let k be the smallest integer such that $a_k \neq \perp$. It follows that, for $k \leq i \leq \ell$, $a_i = (\mathbf{h}_i, x_i)$, where x_i is an appended value, and $\mathbf{h}_i$ is a history returned by a *get-history*() operation on H with linearization point R_i. Then, strictly fewer *append* operations are linearized before R_i than before R. Specifically, step $R_i = \ell$-*buffer-read* is before ℓ-*buffer-write*(a_i), which is the linearization point of an *append*(x_i) operation that is linearized before R.

Whenever $k > 1$, i.e. when *get-history*() operation observes $a_1 = \bot$, it returns $(x_k, x_{k+1}, \cdots, x_\ell)$. There must be only $\ell - k + 1$ occurrences of a *ℓ-buffer-write* step before R. Since each *append* operation is linearized at its *ℓ-buffer-write*, only $\ell - k + 1$ *append* instructions are linearized before R and $x_k, \ldots, x_\ell$ are the values appended by these $\ell - k + 1$ *append* operations, in order from least recent to most recent. Hence, the return value of *get-history* is correct.

Now suppose $k = 1$. Let $\mathbf{h} = \mathbf{h}_m$ be the longest history amongst $\mathbf{h}_1, \ldots, \mathbf{h}_\ell$. If $\mathbf{h}$ contains x_1, then the *get-history*() operation returns $\mathbf{h}' \cdot (x_1, \ldots, x_\ell)$, where $\mathbf{h}'$ is the prefix of $\mathbf{h}$ up to, but not including, x_1. If $\mathbf{h}$ does not contain x_1, then $\mathbf{h} \cdot (x_1, \ldots, x_\ell)$ is returned.

Let W be the *ℓ-buffer-write* step which wrote a_1 to B. There are two cases to consider.

Case 1: $\mathbf{h}$ *contains* x_1. By the induction hypothesis, $\mathbf{h}_m = \mathbf{h}$ contains all values appended to H by *append* operations linearized before step R_m, in order from least recent to most recent. By definition, $x_1, \ldots, x_\ell$ are the last ℓ values appended to H prior to step R, in order. Since $\mathbf{h}$ contains x_1, it must also contain all values appended to H prior to x_1. It follows that $\mathbf{h}' \cdot (x_1, \ldots, x_\ell)$ is the correct return value, where $\mathbf{h}'$ is the prefix of $\mathbf{h}$ up to, but not including, x_1.

Case 2: $\mathbf{h}$ *does not contain* x_1. Let us first show that step $W =$ *ℓ-buffer-write*(a_1), which is the linearization point of *append*(x_1), must happen after R_i for each $1 \leq i \leq \ell$, which is the *get-history*() step by *append*(x_i). Assume for contradiction that W is before R_i. Then, by induction hypothesis, $\mathbf{h}_i$ contains x_1 and also, $\mathbf{h}_i$ is a prefix of $\mathbf{h}$. So $\mathbf{h}$ contains x_1, giving the desired contradiction.

Thus, step W takes place after step R_i for each $2 \leq i \leq \ell$, and before the corresponding *ℓ-buffer-write*(a_i), by definition of ℓ-buffer, x_1 and x_i. Thus, the *append* operations which performed $R_1, \ldots, R_\ell$ are all concurrent. In particular, W occurs in each of their execution intervals. It follows that these *append* operations are all by different processes. Since there are at most ℓ updating processes, there is no *append*

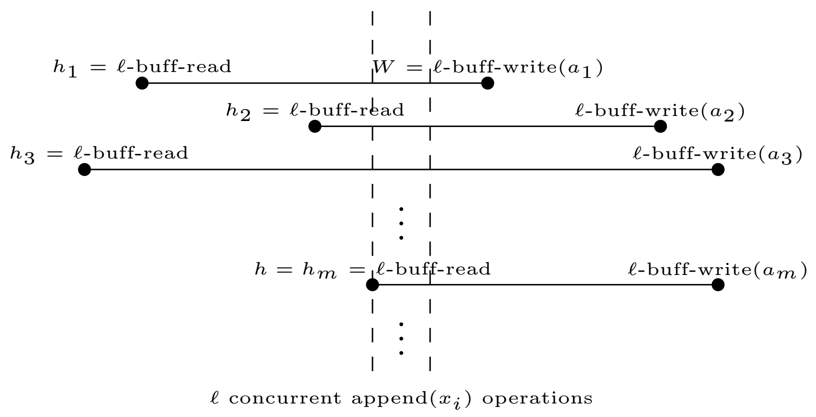


Figure 3-3: Illustration of Case 2 in History object emulation

operation linearized between R_m and W. Therefore, $\mathbf{h}$ contains all values appended to H prior to W. It follows that $\mathbf{h} \cdot (x_1, \ldots, x_\ell)$ is the correct return value. This is illustated in Figure 3-3. $\square$

This lemma allows us to simulate *any* object that supports at most ℓ updating processes using only a single ℓ-buffer. This is because the state of an object is determined by the history of the non-trivial operations performed on it. In particular, we can simulate an array of ℓ single-writer registers using a single ℓ-buffer.

Lemma 3.2.8. *A single ℓ-buffer can simulate ℓ single-writer registers.*

Proof. Suppose that register R_i is owned by process p_i, for $1 \leq i \leq \ell$. By Lemma 3.2.7, it is possible to simulate a history object H that can be updated by ℓ processes and read by any number of processes. To write value x to R_i, process p_i appends (i, x) to H. To read R_i, a process reads H and finds the value of the most recent write to R_i. This is the second component of the last pair in the history whose first component is i. $\square$

Thus, we can use $\lceil \frac{n}{\ell} \rceil$ ℓ-buffers to simulate n single-writer registers. It is well-known that the n-consensus algorithm of Aspnes and Herlihy [AH90] that uses n single-writer registers can be derandomized and made obstruction-free. Therefore, we can simulate this algorithm and solve obstruction-free n-consensus.

Theorem 3.2.9. *It is possible to solve n-consensus using only $\lceil n/\ell \rceil$ ℓ-buffers.*

A Lower Bound

In this section, we show a lower bound on the number of memory locations necessary for solving obstruction-free binary consensus among $n \geq 2$ processes. We will prove this statement for nondeterministic solo terminating protocols [FHS98], which means that from each reachable configuration, for each process, there exists a finite solo execution by the process where it terminates and returns an output. Any algorithm that is obstruction-free satisfies nondeterministic solo termination.

A location r is *covered* by a process p in some configuration, if p is poised to perform *ℓ-buffer-write* on r. An ℓ-buffer is *k-covered* by a set of processes $\mathcal{P}$ in some configuration, if there are exactly k processes in $\mathcal{P}$ that cover it. A configuration is *at most k-covered* by $\mathcal{P}$, if no ℓ-buffer in the system is k'-covered by $\mathcal{P}$ in this configuration, for any $k' > k$.

Let $\mathcal{Q}$ be a set of processes, each of which is poised to perform *ℓ-buffer-write* in some configuration C. A *block write* by $\mathcal{Q}$ *from* C is an execution, starting at C, in which each process in $\mathcal{Q}$ takes exactly one step. If a block write is performed that includes ℓ different *ℓ-buffer-write* instructions to some ℓ-buffer, then any process that performs *ℓ-buffer-read* on that ℓ-buffer immediately afterwards, gets the same result (and ends up in the same state) regardless of the value of that ℓ-buffer in C.

We say that a set of processes $\mathcal{P}$ *can decide* $v \in \{0, 1\}$ *from* a configuration C if there exists a $\mathcal{P}$-only execution from C in which v is decided. If $\mathcal{P}$ can decide both 0 and 1 from C, then $\mathcal{P}$ is *bivalent* from C.

To obtain the lower bound, we extend the proof of the $n-1$ lower bound on the number of registers required for solving n-process consensus [Zhu16], borrowing intuition about reserving executions from the $\Omega(n)$ lower bound for anonymous consensus [Gel15]. The following auxiliary lemmas are largely unchanged from [Zhu16]. The main difference is that we only perform block writes on ℓ-buffers that are ℓ-covered by $\mathcal{P}$.

Lemma 3.2.10. *There is an initial configuration from which the set of all processes in the system is bivalent.*

Proof. Consider an initial configuration I with two processes p_v for $v \in \{0, 1\}$, where p_v starts with input v. Observe that $\{p_v\}$ can decide v from I since, initially, I is indistinguishable from the configuration where every process starts with v to p_v. Thus, $\{p_0, p_1\}$ is bivalent from I and, therefore, so is the set of all processes. $\square$

Lemma 3.2.11. *Let C be a configuration and $\mathcal{Q}$ be a set processes that is bivalent from C. Suppose C is at most ℓ-covered by a set of processes $\mathcal{R}$, where $\mathcal{R} \cap \mathcal{Q} = \varnothing$, and let L be a set of locations that are ℓ-covered by $\mathcal{R}$ in C. Let β be a block write from C by the set of $\ell \cdot |L|$ processes from $\mathcal{R}$ that cover L. Then, there exists a $\mathcal{Q}$-only execution φ from C such that $\mathcal{R} \cup \mathcal{Q}$ is bivalent from $C\varphi\beta$ and in configuration $C\varphi$, some process in $\mathcal{Q}$ covers a location not in L.*

Proof. Suppose some process $p \in \mathcal{R}$ can decide some value $v \in \{0, 1\}$ from configuration $C\beta$ and ϕ is a $\mathcal{Q}$-only execution from C in which $\bar{v}$ is decided. Let φ be the longest prefix of ϕ such that p can decide v from $C\varphi\beta$. Let δ be the next step by $q \in \mathcal{Q}$ in ϕ after φ.

If δ is an *ℓ-buffer-write* to a location in L or is an *ℓ-buffer-read*, then $C\varphi\beta$ and $C\varphi\delta\beta$ are indistinguishable to p. Since p can decide v from $C\varphi\beta$, but p can only decide $\bar{v}$ from $C\varphi\delta\beta$, δ must be an *ℓ-buffer-write* to a location not in L. Thus, in configuration in $C\varphi$, q covers a location not in L, and $C\varphi\beta\delta$ is indistinguishable from $C\varphi\delta\beta$ to process p. Therefore, by definition of φ, p can only decide $\bar{v}$ from $C\varphi\beta\delta$ and p can decide v from $C\varphi\beta$. This implies that $\{p, q\}$ is bivalent from $C\varphi\beta$, as desired. $\square$

The next result says that if a set of processes is bivalent in some configuration, then it is possible to reach a configuration from which some process can decide 0 and some process can decide 1. It does not depend on what instructions are supported by the memory.

Lemma 3.2.12. *Suppose a set $\mathcal{U}$ of at least two processes is bivalent from configuration C. Then it is possible to reach, via a $\mathcal{U}$-only execution from C, a configuration C' such that, a set of at most two processes $\mathcal{Q} \subseteq U$ is bivalent from C'.*

Proof. Let $\mathcal{V}$ be the set of all configurations from which $\mathcal{U}$ is bivalent and which are reachable from C by a $\mathcal{U}$-only execution. Let k be the smallest integer such that there exist a configuration $C' \in \mathcal{V}$ and a set of processes $\mathcal{U}' \subseteq \mathcal{U}$ that is bivalent from C' with $|\mathcal{U}'| = k$. Pick any such $C' \in \mathcal{V}$ and the corresponding set of processes $\mathcal{U}'$.

If $|\mathcal{U}'| = 1$, we set $\mathcal{Q} = \mathcal{U}'$, so suppose $|\mathcal{U}'| \geq 2$. Consider a process $p \in \mathcal{U}'$ and let $\mathcal{U}'' = \mathcal{U}' - \{p\}$ be the set of remaining processes in $\mathcal{U}'$. Since $|\mathcal{U}''| = k - 1$, $\mathcal{U}''$ can only decide v from C' for some $v \in \{0, 1\}$. By nondeterministic solo termination, it follows that each process $q \in \mathcal{U}''$ can decide v from C'.

If p can decide $\bar{v}$ from C' we set $\mathcal{Q} = \{p, q\}$. So, suppose that, like $\mathcal{U}''$, p can only decide v from C'.

Since $\mathcal{U}'$ is bivalent from C', there is a $\mathcal{U}'$-only execution α from C' that decides $\bar{v}$. Let α' be the longest prefix of α such that both p and $\mathcal{U}''$ can only decide v from $C'\alpha'$. Note that $\alpha' \neq \alpha$, because $\bar{v}$ is decided in α. Let δ be the next step in α after α'. Then either p or $\mathcal{U}''$ can decide $\bar{v}$ from $C'\alpha'\delta$.

If δ is a step by a process in $\mathcal{U}''$, then, since $\mathcal{U}''$ can only decide v from $C'\alpha'$, $\mathcal{U}''$ can also only decide v from $C'\alpha'\delta$. Therefore, p can decide $\bar{v}$ from $C'\alpha'\delta$. We are done by setting $\mathcal{Q} = \{p, q\}$, because each process $q \in \mathcal{U}''$ can decide v from $C'\alpha'\delta$.

Finally, suppose that δ is a step by p. Then, since p is can only decide v from $C'\alpha'$, p can also only decide v from $C'\alpha'\delta$. Therefore, $\mathcal{U}''$ can decide $\bar{v}$ from $C'\alpha'\delta$. However, $|\mathcal{U}''| = k - 1$. By definition of k, $\mathcal{U}''$ is not bivalent from $C'\alpha'\delta$. Therefore $\mathcal{U}''$ can only decide $\bar{v}$ from $C'\alpha'\delta$, and hence, every process $q \in \mathcal{U}''$ can decide $\bar{v}$ from $C'\alpha'\delta$. As we know that p can decide v from this configuration, the proof is complete. □

An induction similar to [Zhu16] allows the processes to reach configuration that is at most ℓ-covered by a set of processes $\mathcal{R}$, while another process $z \notin \mathcal{R}$ covers a location that is not ℓ-covered by $\mathcal{R}$. This implies that the configuration is also at most ℓ-covered by $\mathcal{R} \cup \{z\}$, allowing the inductive step to go through.

Lemma 3.2.13. *Let C be a configuration and let $\mathcal{P}$ be a set of $n \geq 2$ processes. If $\mathcal{P}$ is bivalent from C, then there is a $\mathcal{P}$-only execution α and a set of at most two processes $\mathcal{Q} \subseteq \mathcal{P}$ such that $\mathcal{Q}$ is bivalent from $C\alpha$ and $C\alpha$ is at most ℓ-covered by the* *remaining processes $\mathcal{P} - \mathcal{Q}$.*

Proof. By induction on $|\mathcal{P}|$. The base case is when $|\mathcal{P}| = 2$, and holds with the empty execution. Now suppose $|\mathcal{P}| > 2$ and the claim holds for $|\mathcal{P}| - 1$. By Lemma 3.2.12, there is a $\mathcal{P}$-only execution γ and set of at most two processes $\mathcal{Q} \subset \mathcal{P}$ that is bivalent from $D = C\gamma$. Pick any process $z \in \mathcal{P} - \mathcal{Q}$. Then $\mathcal{P} - \{z\}$ is bivalent from D because $\mathcal{Q}$ is bivalent.

We construct a sequence of configurations $D_0, D_1, \ldots$ reachable from D such that, for all $i \geq 0$, the following properties hold:

1. there exists a set of at most two processes $\mathcal{Q}_i \subseteq \mathcal{P} - \{z\}$ such that $\mathcal{Q}_i$ is bivalent from D_i,

2. D_i is at most ℓ-covered by the remaining processes $\mathcal{R}_i = (\mathcal{P} - \{z\}) - \mathcal{Q}_i$,

3. D_{i+1} is reachable from D_i by a $(\mathcal{P} - \{z\})$-only execution α_i which contains a block write β_i to the locations in D_i which are ℓ-covered by processes in $\mathcal{R}_i$.

We can construct D_0 by applying the induction hypothesis to D and $\mathcal{P} - \{z\}$. This gives a $(\mathcal{P} - \{z\})$-only execution η such that the first two properties hold in $D_0 = D\eta$ as required. Now suppose we have D_i, $\mathcal{Q}_i$, $\mathcal{R}_i$ as defined previously. By Lemma 3.2.11 applied to configuration D_i, there is a $\mathcal{Q}_i$-only execution φ_i such that $\mathcal{R}_i \cup \mathcal{Q}_i = \mathcal{P} - \{z\}$ is bivalent from $D_i\varphi_i\beta_i$, where β_i is a block write to the locations in D_i which are ℓ-covered by processes in $\mathcal{R}_i$. Applying the induction hypothesis to $D_i\varphi_i\beta_i$ and $\mathcal{P} - \{z\}$, we get a $(\mathcal{P} - \{z\})$-only execution ψ_i leading to a configuration $D_{i+1} = D_i\varphi_i\beta_i\psi_i$, in which there is a set of at most two processes $\mathcal{Q}_{i+1}$ such that $\mathcal{Q}_{i+1}$ is bivalent from D_{i+1}. Additionally, D_{i+1} is at most ℓ-covered by the set of remaining processes $\mathcal{R}_{i+1} = (\mathcal{P} - \{z\}) - \mathcal{Q}_{i+1}$. Finally, the third property is also satisfied because the execution $\alpha_i = \varphi_i\beta_i\psi_i$ contains the block write β_i.

Since there are only finitely many locations, there exists $0 \leq i < j$ such that $\mathcal{R}_i$ covers the same set of locations in D_i as $\mathcal{R}_j$ does in D_j. We now insert steps of z so that no process in $\mathcal{P} - \{z\}$ can detect them. Consider any $\{z\}$-only execution ζ from $D_i\varphi_i$ that decides a value $v \in \{0, 1\}$. Since $D_i\varphi_i\beta_i$ is bivalent for $\mathcal{P} - \{z\}$, and

β_i block writes to ℓ-covered locations, ζ must contain an *ℓ-buffer-write* to a location that is not ℓ-covered by $\mathcal{R}_i$. Otherwise, $D_i\varphi_i\zeta\beta_i$ is indistinguishable from $D_i\varphi_i\beta_i$ to processes in $\mathcal{P} - \{z\}$, and they can decide $\bar{v}$ from $D_i\zeta\beta_i$, which is impossible. Let ζ' be the longest prefix of ζ containing only writes to locations ℓ-covered by $\mathcal{R}_i$ in D_i. It follows that, in $D_i\varphi_i\zeta'$, z is poised to perform an *ℓ-buffer-write* to a location not ℓ-covered by $\mathcal{R}_i$ in D_i and, hence, $\mathcal{R}_j$ in D_j.

$D_i\varphi_i\zeta'\beta_i$ is indistinguishable from $D_i\varphi_i\beta_i$ to $\mathcal{P} - \{z\}$, so the $(\mathcal{P} - \{z\})$-only execution $\psi_i\alpha_{i+1}\cdots\alpha_{j-1}$ is applicable at $D_i\varphi_i\zeta'\beta_i$. Let $\alpha = \gamma\eta\alpha_0\cdots\alpha_{i-1}\varphi_i\zeta'\beta_i\psi_i\alpha_{i+1}\cdots\alpha_{j-1}$. Every process in $\mathcal{P} - \{z\}$ is in the same state in $C\alpha$ as it is in D_j. In particular, $\mathcal{Q}_j \subseteq \mathcal{P} - \{z\}$ is bivalent from D_j and, hence, from $C\alpha$, and every location in D_j is at most ℓ-covered by $\mathcal{R}_j = (P - \{z\}) - \mathcal{Q}_j$ in D_j and, hence, in $C\alpha$. Moreover, since z takes no steps after $D_i\varphi_i\zeta'$, in $C\alpha$, z covers a location not ℓ-covered by $\mathcal{R}_i$ in D_i and, hence, by $\mathcal{R}_j$ in D_j or $C\alpha$. Therefore, every location is at most ℓ-covered by $\mathcal{R}_j \cup \{z\} = \mathcal{P} - \mathcal{Q}_j$ in $C\alpha$. □

Finally, we can prove the main theorem.

Theorem 3.2.14. *Consider a memory consisting of ℓ-buffers. Then any nondeterministic solo terminating algorithm for solving binary consensus for n processes uses at least $\lceil (n-1)/\ell \rceil$ locations.*

Proof. Consider a nondeterministic solo terminating binary consensus algorithm. Let C be an initial configuration from which the set of all n processes, $\mathcal{P}$, is bivalent. Such a configuration exists by Lemma 3.2.10. Lemma 3.2.13 implies that there is a reachable configuration C and a set of at most two processes $\mathcal{Q} \subseteq \mathcal{P}$ that is bivalent from C. Furthermore, C is at most ℓ-covered by the remaining processes $\mathcal{R} = \mathcal{P} - \mathcal{Q}$. By the pigeonhole principle, $\mathcal{R}$ covers at least $\lceil (n-2)/\ell \rceil$ different locations. If $\lceil (n-2)/\ell \rceil < \lceil (n-1)/\ell \rceil$, then $n-2$ is a multiple of ℓ and every location covered by $\mathcal{R}$ is in fact ℓ-covered by $\mathcal{R}$. By Lemma 3.2.11, since $\mathcal{Q}$ is bivalent from C, we can use a process in $\mathcal{Q}$ to cover a location not ℓ-covered by $\mathcal{R}$. Hence, there are at least $\lceil (n-2)/\ell \rceil + 1 = \lceil (n-1)/\ell \rceil$ locations. □

The lower bound is proven for consensus algorithms that satisfy nondeterministic solo termination. It can also be extended to a *heterogeneous setting*, where memory locations need not be identical, for example, ℓ-buffers for possibly different values of ℓ. For this, we need to extend the definition of at most ℓ-covered to, instead, require each ℓ-buffer to be covered by at most ℓ processes, for all values of ℓ. Then we consider block writes to a set of locations containing ℓ different ℓ*-buffer-write* operations to each ℓ-buffer in the set. The general result is that, for any algorithm which solves consensus for n processes and satisfies nondeterministic solo termination, the sum of capacities of all buffers must be at least $n-1$.

The lower bound also applies to systems in which the return value of every non-trivial instruction on a memory location does not depend on the value of that location and the return value of any trivial instruction is a function of the sequence of the preceding ℓ non-trivial instructions performed on the location. This is because such instructions can be implemented by ℓ*-buffer-read* and ℓ*-buffer-write* instructions. We record each invocation of a non-trivial instruction using ℓ*-buffer-write*. The return values of these instructions can be determined without even looking at the memory. To implement a trivial instruction, we perform ℓ*-buffer-read*, which returns a sequence containing the description of the last ℓ non-trivial instructions performed on the location, which is sufficient to determine the correct return value.

3.2.5 Multiple Assignment

With m-register multiple assignment, we can atomically write to m locations. This plays an important role in [Her91], as m-register multiple assignment can used to solve wait-free consensus for $2m-2$ processes, but not for $2m-1$ processes.

In this section, we explore whether multiple assignment could improve the space complexity of solving obstruction-free consensus. A practical motivation to this question is that obstruction-free multiple assignment can be easily implemented using a simple transaction.

We prove a lower bound that is similar to the lower bound in Section 3.2.4. Suppose ℓ*-buffer-read*$()$ and ℓ*-buffer-write*(x) instructions are supported on every memory location in a system and, for any subset of locations, we are allowed to atomically perform one *ℓ-buffer-write* instruction per location. Then $\lceil n/2\ell \rceil$ locations are necessary for n processes to solve nondeterministic solo terminating consensus. As in Section 3.2.4, this result can be further generalized to different sets of instructions, and heterogeneous settings.

The main technical difficulty is proving an analogue of Lemma 3.2.11. In the absence of multiple assignment, if β is a block write to a set of ℓ-covered locations L and δ is an *ℓ-buffer-write* to a location not in L, then β and δ trivially commute (in the sense that the resulting configurations are indistinguishable to all processes). However, a multiple assignment δ may now atomically *ℓ-buffer-write* to many locations, including locations in L. Thus, it is now possible for processes to distinguish between $\beta\delta$ and $\delta\beta$. Using a careful combinatorial argument, we show how to perform two blocks of multiple assignments β^1 and β^2 such that, in β^i for $i \in \{0, 1\}$, *ℓ-buffer-write* is performed at least ℓ times on each location in L, and is never performed on any location not in L. Given this, we can show that $\beta^1\delta\beta^2$ and $\delta\beta^1\beta^2$ are indistinguishable to all processes, which is enough to prove an analogue of Lemma 3.2.11.

First, we define the notion of covering in this setting. We say that a process p *covers* location r in a configuration C, when p is poised to perform a multiple assignment in C that involves an *ℓ-buffer-write* to r (recall that in our definition of *poised*, we always consider a single, fixed next step of each process). The next definition is key to our proof. *A k-packing* of a set of processes $\mathcal{P}$ in some configuration C is a function π mapping each process in $\mathcal{P}$ to some memory location it covers such that no location r has more than k processes mapped to it (i.e., $|\pi^{-1}(r)| \leq k$). When $\pi(p) = r$ we say that π *packs* p *in* r. A k-packing may not always exist, or there may be many, depending on the configuration, the set of processes, and the value of k. A location r is *fully k-packed* by $\mathcal{P}$ in configuration C, if a k-packing of $\mathcal{P}$ exists in C and all such k-packings pack exactly k processes in r.

Suppose that, in some configuration, there are two k-packings of the same set of processes such that the first packs more processes in some location r than the second. We show there is a location r' in which the first packing packs fewer processes than the second and a k-packing that pack one less process to location r, one more process to location r' and the same number of processes to all other locations, as compared to the first packing. This iterative procedure has some resemblance to cuckoo hashing [PR01]. The proof relies on existence of a certain Eulerian path in a multigraph that we build to represent these two k-packings.

Lemma 3.2.15. *Suppose g and h are two k-packings of the same set of processes $\mathcal{P}$ in some configuration C and r_1 is a location such that g packs more processes in r_1 than h does (i.e., $|g^{-1}(r_1)| > |h^{-1}(r_1)|$). Then, there exists a sequence of locations $r_1, r_2, \ldots, r_t$ and a sequence of distinct processes $p_1, p_2, \ldots, p_{t-1}$, such that h packs more processes in r_t than g, (i.e., $|h^{-1}(r_t)| > |g^{-1}(r_t)|$) and, for $1 \leq i \leq t-1$, $g(p_i) = r_i$ and $h(p_i) = r_{i+1}$.*

Proof. Consider a multigraph with one node for each memory location in the system and one directed edge labelled by p, from node $g(p)$ to node $h(p)$, for each process $p \in \mathcal{P}$. Therefore, the in-degree of a node v is equal to $|h^{-1}(v)|$, which is the number of processes that are packed into memory location v by h, and the out-degree of node v is equal to $|g^{-1}(v)|$, which is the number of processes that are packed in v by g.

Now, consider any maximal Eulerian path in this multigraph starting from the node r_1. This path follows a sequence of edges for as long as possible without repeating any edge. Let $r_1, \ldots, r_t$ be the sequence of nodes visited (which may contain repetitions) and let p_i be the labels of the traversed edges, in order. Then $g(p_i) = r_i$ and $h(p_i) = r_{i+1}$ for $1 \leq i \leq t-1$. Since the path is Eulerian, the labels of the edges are distinct, which, in turn, guarantees that the sequence is finite. Finally, by maximality, the last node in the sequence must have more incoming than outgoing edges, so $|h^{-1}(r_t)| > |g^{-1}(r_t)|$. $\square$

Corollary 3.2.16. *Let the k-packings g and h and the sequences r_i and p_i be defined as in Lemma 3.2.15. For $1 \leq j < t$, there exists a k-packing g', such that g' packs one less process than g in r_j, one more process than g in r_t, and the same number of processes as g in all other locations.*

Proof. We construct g' from g by re-packing each process p_i from r_i to r_{i+1} for all $j \leq i < t$. Then $g'(p_i) = r_{i+1}$ for $j \leq i < t$ and $g'(p) = g(p)$ for all other processes p. Notice that p_i covers r_{i+1}, since $h(p_i) = r_{i+1}$ and h is a k-packing.

As compared to g, g' packs one less process in r_j, one more process in r_t, and the same number of processes in every other location. Since h is a k-packing, it packs at most k processes in r_t. Because g is a k-packing that packs less processes in r_t than h, g' is also a k-packing. □

Let $\mathcal{P}$ be a set of processes, each of which is poised to perform a multiple assignment in some configuration C. A *block multi-assignment* by $\mathcal{P}$ from C is an execution starting at C, in which each process in $\mathcal{P}$ takes exactly one step.

Consider some configuration C, a set of processes $\mathcal{R}$, such that a 2ℓ-packing π of $\mathcal{R}$ in C exists. Let L be the set of all locations that are fully 2ℓ-packed by $\mathcal{R}$ in C. By definition, π packs exactly 2ℓ processes from $\mathcal{R}$ in each location $r \in L$. Let us partition these $2\ell \cdot |L|$ processes packed in L by π into two sets $\mathcal{R}^1, \mathcal{R}^2 \subseteq \mathcal{R}$ of $\ell \cdot |L|$ processes each such that, for each location $r \in L$, exactly ℓ packed processes belong to $\mathcal{R}^1$ and the remaining ℓ packed processes belong to $\mathcal{R}^2$. Let β_i be a block multi-assignment by $\mathcal{R}^i$ for $i \in \{1, 2\}$.

Notice that, for any location $r \in L$, after β_i, the outcome of any subsequent *ℓ-buffer-read* on r does not depend on multiple assignments that occurred prior to the block multi-assignment. Moreover, we can prove the following crucial property about these block multi-assignments to fully packed locations.

Lemma 3.2.17. *Neither β_1 nor β_2 involves an ℓ-buffer-write to a location outside of L.*

Proof. Assume the contrary. Let $q \in \mathcal{R}^1 \cup \mathcal{R}^2$ be a process with $\pi(q) \in L$ such that q also covers some location $r_1 \notin L$ in C. It must be the case that π packs exactly 2ℓ processes in r_1, i.e. $|\pi^{-1}(r_1)| = 2\ell$. Otherwise, re-packing q in r_1 instead of $\pi(q)$ leads to another 2ℓ-packing that does not pack exactly 2ℓ processes in $\pi(q) \in L$, contradicting the definition of a fully 2ℓ-packed location.

Since L is the set of all fully 2ℓ-packed locations, there exists a 2ℓ-packing h, which packs strictly less than 2ℓ processes in $r_1 \notin L$. Applying Lemma 3.2.15 for 2ℓ-packings π and h, we get a sequence of locations $r_1, \ldots, r_t$ and a sequence of processes $p_1, \ldots p_{t-1}$. $|\pi^{-1}(r_t)| < |h^{-1}(r_t)|$, and since h is a 2ℓ-packing, we must have that $|\pi^{-1}(r_t)| < 2\ell$, i.e. π packs strictly less than 2ℓ processes in r_t in C. We consider two cases.

First, suppose that q does not occur in the sequence, i.e. $\forall i : p_i \neq q$. We know that $|\pi^{-1}(r_1)| = 2\ell$ and $|\pi^{-1}(r_t)| < 2\ell$, implying that the locations r_1 and r_t must be different. This allows us to apply Corollary 3.2.16 for the whole sequence with $j = 1$. We get a 2ℓ-packing π' that packs less than 2ℓ processes in r_1, and exactly 2ℓ processes in each of the fully 2ℓ-packed locations L, including $\pi(q)$. Moreover, we did not re-pack process q, so $\pi'(q) = \pi(q)$. Hence, modifying π' by re-packing q in r_1 instead of $\pi'(q)$ again leads to a 2ℓ-packing that does not pack exactly 2ℓ processes in a location in L, a contradiction.

Now, assume that $q = p_s$, for some s. By the properties of our sequences, we know $r_s = \pi(p_s) = \pi(q) \in L$, and since this location is fully 2ℓ-packed in configuration C we have $|\pi^{-1}(r_s)| = 2\ell$. Similar to the first case, since $|\pi^{-1}(r_t)| < 2\ell$ in C, locations r_s and r_t must be different and we now apply Corollary 3.2.16 with $j = s$. We get a 2ℓ-packing π'' that packs less than 2ℓ processes in location $r_s \in L$. This is a contradiction with the definition of a fully 2ℓ-packed location, completing the proof. □

We can now prove a lemma that replaces Lemma 3.2.11 in the main argument.

Lemma 3.2.18. *Let C be a configuration and $\mathcal{Q}$ be a set of processes that is bivalent from C. Suppose there is a set of processes $\mathcal{R}$ disjoint from $\mathcal{Q}$ such that there exists a 2ℓ-packing π of $\mathcal{R}$ in C. Let L be the set of fully 2ℓ-packed locations by $\mathcal{R}$ in C. Consider block multi-assignments β^1 and β^2 from C, as defined above. Then, there exists a $\mathcal{Q}$-only execution φ from C such that $\mathcal{R} \cup \mathcal{Q}$ is bivalent from $C\varphi\beta^1$ and in configuration $C\varphi$, some process in $\mathcal{Q}$ covers a location not in L.*

Proof. Suppose some process $p \in \mathcal{R}$ can decide a value $v \in \{0, 1\}$ from configuration $C\beta^1\beta^2$ and ϕ is a $\mathcal{Q}$-only execution from C in which $\bar{v}$ is decided. Let φ be the longest prefix of ϕ such that p can decide v from $C\phi\beta^1\beta^2$. Let δ be the next step by $q \in \mathcal{Q}$ in ϕ after φ.

Since p can decide v from $C\varphi\beta^1\beta^2$, but p can only decide $\bar{v}$ from $C\varphi\delta\beta^1\beta^2$, δ must be a multiple assignment which includes an *ℓ-buffer-write* to a location not in L. If δ was a read, the resulting configurations would be indistinguishable. Similarly, if δ was a multiple assignment involving only *ℓ-buffer-write*'s to locations in L, then no process in $\mathcal{R}$ would be able to observe a difference due to the block multi-assignments $\beta^1\beta^2$. Thus, in configuration $C\varphi$, q covers a location not in L, as desired.

We claim that the configuration $C\varphi\beta^1\delta\beta^2$ is indistinguishable from $C\varphi\delta\beta^1\beta^2$ to the process p. Indeed, for each location $r \in L$, the contents of location r are the same in $C\varphi\delta\beta^1\beta^2$ as it is in $C\varphi\beta^1\delta\beta^2$ due to the block multi-assignment β^2. On the other hand, for each location $r \notin L$, by Lemma 3.2.17, neither β^1 nor β^2 performs an *ℓ-buffer-write* to r, thus, the contents of r are the same in $C\varphi\delta\beta^1\beta^2$ and in $C\varphi\beta^1\delta\beta^2$. Finally, the state of process p is the same in both of these configurations.

Therefore, p can only decide $\bar{v}$ from $C\varphi\beta^1\delta\beta^2$ and hence, $C\varphi\beta^1$ is $\bar{v}$-deciding for $\mathcal{R} \cup q$. Moreover, p can decide v from $C\varphi\beta^1\beta^2$ by definition of φ, and thus $C\varphi\beta^1$ is v-deciding for $\mathcal{R}$. We have established the desired bivalency of $\mathcal{R} \cup \mathcal{Q}$ from $C\varphi\beta^1$, completing the proof. $\square$

Using these tools, we can prove the following analogue of Lemma 3.2.13:

Lemma 3.2.19. *Let C be a configuration and let $\mathcal{P}$ be a set of $n \geq 2$ processes. If $\mathcal{P}$ is bivalent from C, then there is a $\mathcal{P}$-only execution α and a set of at most two processes $\mathcal{Q} \subseteq \mathcal{P}$ such that $\mathcal{Q}$ is bivalent from $C\alpha$ and there exists a 2ℓ-packing π of the remaining processes $\mathcal{P} - \mathcal{Q}$ in $C\alpha$.*

Proof. By induction on $|\mathcal{P}|$. The base case is when $|\mathcal{P}| = 2$, and holds with the empty execution. Now suppose $|\mathcal{P}| > 2$ and the claim holds for $|\mathcal{P}| - 1$. By Lemma 3.2.12, there is a $\mathcal{P}$-only execution γ and a set of at most two processes $\mathcal{Q} \subset \mathcal{P}$ that is bivalent from $D = C\gamma$. Pick any process $z \in \mathcal{P} - \{p_0, p_1\}$. Then $\mathcal{P} - \{z\}$ is bivalent from D because $\mathcal{Q}$ is bivalent.

We construct a sequence of configurations $D_0, D_1, \ldots$ reachable from D, such that, for all $i \geq 0$, the following properties hold:

1. there exists a set of at most two processes $\mathcal{Q}_i \subseteq \mathcal{P} - \{z\}$ such that $\mathcal{Q}_i$ is bivalent from D_i,
2. there exists a 2ℓ-packing π_i of the remaining processes $\mathcal{R}_i = (\mathcal{P} - \{z\}) - \mathcal{Q}_i$ in D_i,
3. D_{i+1} is reachable from D_i by a $(\mathcal{P} - \{z\})$-only execution α_i, and
4. Let L_i be the set of all fully 2ℓ-packed locations by $\mathcal{R}_i$ in D_i. Execution α_i contains a block multi-assignment β_i such that, for each location $r \in L_i$, β_i involves at least ℓ multiple assignments which perform *ℓ-buffer-write* on r.

We can construct D_0 by applying the induction hypothesis to D and $\mathcal{P} - \{z\}$. This gives a $(\mathcal{P} - \{z\})$-only execution η such that the first two properties hold in $D_0 = D\eta$ as required. Now suppose we have D_i, $\mathcal{Q}_i$, $\mathcal{R}_i$, π_i, and L_i as defined previously. By Lemma 3.2.18 applied to configuration D_i, there is a $\mathcal{Q}_i$-only execution φ_i such that $\mathcal{R}_i \cup \mathcal{Q}_i = \mathcal{P} - \{z\}$ is bivalent from $D_i\varphi_i\beta_i$, and where $\beta_i = \beta^1$ is a block multi-assignment in which for each location $r \in L_i$, *ℓ-buffer-write* is performed at least ℓ times on r, as desired. Applying the induction hypothesis to $D_i\varphi_i\beta_i$ and $\mathcal{P} - \{z\}$, we get a $(\mathcal{P} - \{z\})$-only execution ψ_i leading to a configuration $D_{i+1} = D_i\varphi_i\beta_i\psi_i$, in which there is a set of at most two processes $\mathcal{Q}_{i+1}$ such that $\mathcal{Q}_{i+1}$ is bivalent from D_{i+1}. Additionally, there exists a 2ℓ-packing π_i of the remaining processes $\mathcal{R}_{i+1} = (\mathcal{P} - \{z\}) - \mathcal{Q}_{i+1}$ in D_{i+1}. Finally, the third and fourth properties are also satisfied as the execution $\alpha_i = \varphi_i\beta_i\psi_i$ contains the block multi-assignment β_i.

Since there are only finitely many locations, there exists $0 \leq i < j$ such that $L_i = L_j$. That is, the set of fully 2ℓ-packed locations by $\mathcal{R}_i$ in D_i is the same as the set of fully 2ℓ-packed locations by $\mathcal{R}_j$ in D_j. Let $L = L_i$. We now insert steps of z so that no process in $\mathcal{P} - \{z\}$ can detect them. Consider any $\{z\}$-only execution ζ from $D_i\varphi_i$ that decides a value $v \in \{0, 1\}$. Since $D_i\varphi_i\beta_i$ is bivalent for $\mathcal{P} - \{z\}$, and, for each location $r \in L$, β_i contains at least ℓ multiple assignments which perform

ℓ-buffer-write to r, ζ must contain an *ℓ-buffer-write* to a location not in L. Otherwise, $D_i\varphi_i\zeta\beta_i$ is indistinguishable from $D_i\varphi_i\beta_i$ to processes in $\mathcal{P}-\{z\}$, and they can decide $\overline{v}$ from $D_i\zeta\beta_i$, which is impossible. Let ζ' be the longest prefix of ζ containing only multiple assignments that involve writes to locations in L. It follows that, in $D_i\varphi_i\zeta'$, z is poised to perform an *ℓ-buffer-write* to a location not in $L = L_i$, and hence, not in L_j.

$D_i\varphi_i\zeta'\beta_i$ is indistinguishable from $D_i\varphi_i\beta_i$ to $\mathcal{P}-\{z\}$, so the $(\mathcal{P}-\{z\})$-only execution $\psi_i\alpha_{i+1}\cdots\alpha_{j-1}$ is applicable at $D_i\varphi_i\zeta'\beta_i$. Let $\alpha = \gamma\eta\alpha_0\cdots\alpha_{i-1}\varphi_i\zeta'\beta_i\psi_i\alpha_{i+1}\cdots\alpha_{j-1}$. Every process in $\mathcal{P}-\{z\}$ is in the same state in $C\alpha$ as it is in D_j. In particular, $\mathcal{Q}_j \subseteq \mathcal{P}-\{z\}$ is bivalent from D_j and, hence, from $C\alpha$. Furthermore, the 2ℓ-packing π_j of $\mathcal{R}_j$ in D_j is a 2ℓ-packing of $\mathcal{R}_j$ in $C\alpha$, and the set of locations that are fully 2ℓ-packed by $\mathcal{R}_j$ in $C\alpha$ is L. Since z takes no steps after $D_i\varphi\zeta'$, in $C\alpha$, z covers a location r not in L. By definition of L, since $r \notin L$, there is a 2ℓ-packing π'_j of $\mathcal{R}_j$ in $C\alpha$ which packs less than 2ℓ processes into r. Thus, we can define a 2ℓ-packing π of $\mathcal{R}_j \cup \{z\}$ by packing each process in $\mathcal{R}_j$ according to π'_j and packing z into r. It follows that π is a 2ℓ-packing of $\mathcal{P}-\mathcal{Q}_j = \mathcal{R}_j \cup \{z\}$ in $C\alpha$. □

We can now prove the main theorem.

Theorem 3.2.20. *Consider a memory consisting of ℓ-buffers. If the processes can atomically ℓ-buffer-write to any subset of the ℓ-buffers, then any algorithm for solving nondeterministic solo terminating consensus for n processes uses at least $\lceil (n-1)/2\ell \rceil$ locations.*

Proof. Consider a binary consensus algorithm satisfying nondeterministic solo termination. Let C be an initial configuration from which the set of all n processes, $\mathcal{P}$, is bivalent. Such a configuration exists by Lemma 3.2.10. Lemma 3.2.19 implies that there is a reachable configuration C and a set of at most two processes $\mathcal{Q} \subseteq \mathcal{P}$ that is bivalent from C. Furthermore, there exists a 2ℓ-packing π of the remaining processes $\mathcal{R} = \mathcal{P}-\mathcal{Q}$ in C. By the pigeonhole principle, $\mathcal{R}$ covers at least $\lceil (n-2)/2\ell \rceil$ different locations. If $\lceil (n-2)/2\ell \rceil < \lceil (n-1)/2\ell \rceil$, then $n-2$ is a multiple of 2ℓ, and every location is fully 2ℓ-packed by $\mathcal{R}$. By Lemma 3.2.18, since $\mathcal{Q}$ is bivalent from C, we can use a process in $\mathcal{Q}$ to cover a location not fully 2ℓ-packed by $\mathcal{R}$. Hence, there are at least $\lceil (n-2)/2\ell \rceil + 1 = \lceil (n-1)/2\ell \rceil$ locations. □

3.3 Universality using "Weak" Instructions

In order to develop efficient concurrent algorithms and data-structures in multiprocessor systems, processes that take steps asynchronously need to coordinate their actions. In shared memory systems, this is accomplished by applying hardware-supported low-level atomic instructions to memory locations. An atomic instruction takes effect as a single indivisible step. The most natural and universally supported instructions are *read* and *write*, as these are useful even in uniprocessors to store and load data from memory.

A concurrent implementation is *wait-free*, if any process that takes infinitely many steps completes infinitely many operation invocations. An implementation is *lock-free* if in any infinite execution infinitely many operations are completed. The celebrated FLP impossibility result [FLP85] implies that in a system equipped with only *read* and *write* instructions, there is no deterministic algorithm to solve binary lock-free/wait-free consensus among $n \geq 2$ processes. Binary consensus is a synchronization task where processes start with input bits, and must agree on an output bit that was an input to one of the processes. For one-shot tasks like consensus, wait-freedom and lock-freedom are equivalent.

Herlihy's Consensus Hierarchy [Her91] takes the FLP result further. It assigns a *consensus number* to each object, namely, the number of processes for which there is a wait-free binary consensus algorithm using only instances of this object and *read-write* registers. An object with a higher consensus number is hence a more powerful tool for synchronization. Moreover, Herlihy showed that consensus is a fundamental synchronization task, by developing a universal construction which allows n processes to wait-free implement any object with a sequential specification, provided that they can solve consensus among themselves.

Herlihy's hierarchy is simple, elegant and, for many years, has been our best explanation of synchronization power. It provides an intuitive explanation as to why, for instance, the *compare-and-swap* instuction can be viewed "stronger" than *fetch-and-increment*, as the consensus number of a Compare-and-Swap object is n, while the consensus number of Fetch-and-Increment is 2.

However, key to this hierarchy is treating synchronization instructions as distinct objects, an approach that is far from the real-world, where multiprocessors do let processes apply supported atomic instructions to arbitrary memory locations. In fact, a recent work by Ellen et al. [EGSZ16] has shown that a combination of instructions like *decrement* and *multiply-by-n*, whose corresponding objects have consensus number 1 in Herlihy's hierarchy, when applied to the same memory location, allows solving wait-free consensus for n processes. Thus, in terms of computability, a combination of instructions traditionally viewed as "weak" can be as powerful as a *compare-and-swap* instruction, for instance.

The practical question is whether we can really replace a *compare-and-swap* instruction in concurrent algorithms and data-structures with a combination of weaker instructions. This might seem improbable for two reasons. First, *compare-and-swap* is ubiquitous in practice and used heavily for various tasks like swinging a pointer. Second, the protocol given by Ellen et al. solves only binary n-process consensus. It is not clear how to use it for implementing complex concurrent objects, as utilizing Herlihy's universal construction is not a practical solution. On the optimistic side, there exists a concurrent queue implementation based on *fetch-and-add* that outperforms *compare-and-swap*-based alternatives [MA13]. Both a Queue and a Fetch-and-Add object have consensus number 2, and this construction does not "circumvent" Herlihy's hierarchy by applying different non-trivial synchronization instructions to the same location. Indeed, we are not aware of any practical construction that relies on applying different instructions to the same location.

As a proof of concept, we develop a lock-free universal construction using only *read*, *xor*, *decrement*, and *fetch-and-increment* instructions. The construction could be made wait-free by standard helping techniques. In particular, we implement a Log object [BMW$^+$13] (also known as a History object [Dav04]), which supports high-level operations *get-log*() and *append*(*item*), and is linearizable [HW90] to the sequential specification that *get-log*() returns all previously appended items in order. This interface can be used to agree on a simulated object state, and thus, provides the universal construction [Her91]. In practice, we require a *get-log*() for each thread to return a suffix of items after the last *get-log*() by this thread. We design a lock-free Log with wait-free readers, which performs as well as a *compare-and-swap*-based solution on modern hardware.

In our construction, we could replace both *fetch-and-increment* and *decrement* with the atomic *fetch-and-add* instruction, reducing the instruction set size even further.

3.3.1 Algorithm

We work in the bounded concurrency model where at most n processes will ever access the Log implementation. The object is implemented by a single *fetch-and-increment*-based counter C, and an array A of b-bit integers on which the hardware supports atomic *xor* and *decrement* instructions. We assume that A is unbounded. Otherwise, processes can use A to agree on the next array A' to continue the construction. C and the elements of A are initialized by 0. We call an array location *invalid* if it contains a negative value, i.e., if its most significant bit is 1, *empty* if it contains value 0, and *valid* otherwise. The least significant $m = \lceil \log_2 (n+1) \rceil$ bits are *contention bits* and have a special importance to the algorithm. The remaining $b - m - 1$ bits are used to store items. See Figure 3-4 for illustration.

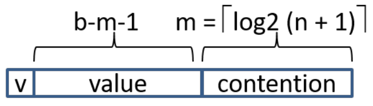


Figure 3-4: Element of A.

For every array location, at most one process will ever attempt to record a $(b - m - 1)$-bit item, and at most $n - 1$ processes will attempt to invalidate this location. No process will try to record to or invalidate the same location twice. In order to record item x, a process invokes $xor(x')$, where x' is x shifted by m bits to the left, plus $2^m - 1 \geq n$, i.e., the contention bits set to 1. To invalidate a location, a process calls a *decrement*. The following properties hold:

1. After a *xor* or *decrement* is performed on a location, no *read* on it ever returns 0.

2. If a *xor* is performed first, no later read returns an invalid value. Ignoring the most significant bit, the next most significant $b - m - 1$ bits contain the item recorded by *xor*.

3. If a *decrement* is performed first, then all values returned by later *read*s are invalid.

A *xor* instruction *fails to record an item* if it is performed after a decrement.

To implement a *get-log*() operation, process p starts at index $i = 0$, and keeps reading the values of $A[i]$ and incrementing i until it encounters an empty location $A[i] = 0$. By the above properties, from every valid location $A[j]$, it can extract the item x_j recorded by a *xor*, and it returns an ordered list of all such items $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$. In practice, we require p to return only a suffix of items appended after the last *get-log*() invocation by p. This can be accomplished by keeping i in static memory instead of initializing it to 0 in every invocation. To make *get-log* wait-free, p first performs $l = C.read()$. Then, if i becomes equal to l during the traversal, it stops and returns the items extracted so far.

To implement *append*(x), process p starts by $\ell = C.\textit{fetch-and-increment}()$. Then it attempts to record item x in $A[\ell]$ using an atomic *xor* instruction. If it fails to record an item, the process does another *fetch-and-increment* and attempts *xor* at that location, and so on, until it is able to successfully record x. Suppose this location is $A[\ell']$. Then p iterates from $j = \ell' - 1$ down to $j = 0$, reading each $A[j]$, and if $A[j]$ is empty, performing a *decrement* on it. Afterwards, process p can safely return.

fetch-and-increment guarantees that each location is *xor*ed at most once, and it can be *decrement*ed at most $n - 1$ times, once by each process that did not *xor*. As a practical optimization, each process can store the maximum ℓ' from its previous *append* operations and only iterate down to ℓ' in the next invocation (all locations with lower indices will be non-empty). Our implementation of *append* is lock-free, because if an operation takes steps and does not terminate it must be repeatedly failing to record items in locations. This only happens if other *xor* operations successfully record their items and invalidate these locations.

At any time t during the execution, let us denote by $f(t)$ as the maximum index such that, $A[f(t)]$ is valid and $A[j]$ is non-empty for all $j \leq f(t)$. By the first property $f(t)$ is non-decreasing, i.e., for $t' > t$ we have $f(t') \geq f(t)$. We linearize an $\mathit{append}(x)$ operation by p that records x at location $A[\ell]$ at the smallest t where $f(t) \geq \ell$. This happens during the operation by p, as when p starts $\mathit{append}(x)$, $A[\ell]$ is empty, and when p finishes, $A[0] \neq 0, \ldots, A[\ell - 1] \neq 0$ and $A[\ell]$ is valid. Next, we show how to linearize *get-log*().

Consider a *get-log*() operation with the latest returned item x_ℓ extracted from $A[\ell]$. We show by contradiction that the execution interval of this *get-log*() must contain time t such that $f(t) = \ell$. We then linearize *get-log*() at the smallest such t. It is an easy exercise to deal with the case when multiple operations are linearized at exactly the same point by slightly perturbing linerization points to enforce the correct ordering. Suppose the *get-log*() operation extracts x_ℓ from $A[\ell]$ at time T. $f(T) \geq \ell$ as *get-log*() stops at an empty index, and by the contradiction assumption we must have $\ell' = f(T) > \ell$. *get-log*() then reaches valid location $A[\ell']$ and extracts an item $x_{\ell'}$ from it, contradicting the definition of x_ℓ.

We implemented the algorithm on X86 processor and with 32 threads. It gave the same performance as an implementation that used *compare-and-swap* for recording items and invalidating locations. It turns out that in today's architectures, the cost of supporting *compare-and-swap* is not significantly higher than that of supporting *xor* or *decrement*. This may or may not be the case in future Processing-in-Memory architectures [PAC+97]. Finding a compact set of synchronization instructions that, when supported, is equally powerful as the set of instructions used today is an important question to establish in future research.

3.4 Enter the Simulation: k-Set Agreement

3.4.1 Model

We consider the standard asynchronous shared memory system with n processes that communicate using shared multi-reader, multi-writer registers. Without loss of generality [AAD+93], we consider protocols that use a multi-writer atomic snapshot where processes alternately update and scan.

We are interested in the space complexity of x-obstruction-free protocols for the *k-set agreement problem*. In this problem, each process starts with an input in $\{0, 1, \ldots, k\}$ and must decide a value that is the input of some process such that the set of decided values among all processes has size at most k.

A protocol satisfies *x-obstruction-freedom* if, for any configuration C and for any set $\mathcal{P}$ of at most x processes, an infinite $\mathcal{P}$-only execution does not exist. In other words, if only processes in $\mathcal{P}$ take steps and they take sufficiently many steps, then each process that takes steps has to decide a value and terminate. In this work, we will be focusing on the k-obstruction-free setting.

We call an execution *$\mathcal{P}$-only* for a set of processes $\mathcal{P}$, if it contains only steps by processes in $\mathcal{P}$. A *solo execution* of process p is a $\{p\}$-only execution. We say that a process p *decides* a value v when it returns v. We say that a set of processes $\mathcal{P}$ *can decide* value v from configuration C or has *valency* v in configuration C when there exists a possibly empty $\mathcal{P}$-only execution α from C such that some process $p \in \mathcal{P}$ has decided v in $C\alpha$. We say that process p *can decide* v from configuration C or has *valency* v in configuration C, when $\{p\}$ does.

We will rely on the following impossibility result for k-set agreement.

Theorem 3.4.1. *Let $k \geq 1$. Suppose $k+1$ processes start with $k+1$ different inputs. There is no deterministic wait-free protocol that allows the processes to collectively output at most k different input values subject to condition that a value v is output only if the process with input value v has taken at least one step.*

3.4.2 Local Argument

Boundary Condition: A protocol for k-set agreement satisfies the *boundary condition* if, for any set of processes $\mathcal{P}$ in any reachable configuration C, the following holds. Let V be the set of all valencies of $\mathcal{P}$ in C. Then, there exists a (possibly empty) $\mathcal{P}$-only execution α from C such that

1. the set of all valencies of $\mathcal{P}$ in $C\alpha$ is still V, and
2. for any $\mathcal{Q} \subseteq \mathcal{P}$ and any valency v of $\mathcal{Q}$ in $C\alpha$, there exists $q \in \mathcal{Q}$ with valency v in $C\alpha$.

The motivation behind the boundary condition is that it lets us reach configurations that look like initial configurations, which should allow us to derive a "local" FLP-style space lower bound by inductively extending an execution. This is exactly what we will do in the rest of this section.

Lemma 3.4.2 (Baby Simulation 1). *Consider a k-set agreement protocol, that satisfies the boundary condition and k-obstruction-freedom. If a set of processes $\mathcal{P}$ has valencies $\{0, 1, \ldots, k\}$ in some configuration C, then no process has already decided in C.*

Proof. Assume the contrary. Without loss of generality, suppose a process p has decided value k in C. Since $\mathcal{P}$ has valencies $\{0, 1, \ldots, k\}$ in C and the protocol satisfies the boundary condition, there exists a $\mathcal{P}$-only execution α and there exists $p_i \in \mathcal{P}$, for each $i \in \{0, 1, \ldots, k-1\}$, such that p_i has valency i in $C\alpha$. If there exists a $\mathcal{P}$-only execution α' from $C\alpha$ such that every value in $\{0, \ldots k-1\}$ has been decided in $C\alpha\alpha'$, then k-set agreement is violated. This and the valency of p_0 in $C\alpha$ immediately gives a contradiction for $k = 1$.

For $k \geq 2$, we construct a wait-free protocol for the $(k-1)$-agreement task among k processes, contradicting the known impossibility result stated in Theorem 3.4.1. The idea is that process q_i with input i in the wait-free protocol simulates processes p_i starting in configuration $C\alpha$. If p_i has already decided i in $C\alpha$, q_i immediately returns i. By the boundary condition, for any set of values $V \subseteq \{0, 1 \ldots, k-1\}$, a

$\{p_v : v \in V\}$-only execution from $C\alpha$ can only decide values from V. Thus, simulating processes can only decide a value v in executions where q_v has taken at least one step, as required. The simulated protocol is k-obstruction-free, $|\{q_0, \dots q_{k-1}\}| = k$, and each q_i simulates a single process p_i, every simulation is wait-free. We designed a wait-free protocol for k processes to agree on $k-1$ values, completing the proof by contradiction. $\square$

The next lemma is also proved using a simulation, albeit a more complex one.

Lemma 3.4.3 (Baby Simulation 2)**.** *Consider a k-set agreement protocol, that satisfies the boundary condition and k-obstruction-freedom. Suppose C is a configuration in which a set of processes $\mathcal{R}$ covers $|\mathcal{R}|$ registers. If there is a set of processes $\mathcal{P}$ disjoint from $\mathcal{R}$ that has valencies $\{0, 1, \dots, k\}$ in C, then there is a $\mathcal{P}$-only execution γ starting from C such that P still has valencies $\{0, 1, \dots, k\}$ in $C\gamma\beta$, where β is the block write by $\mathcal{R}$.*

Proof. Assume the contrary. Let α be the $\mathcal{P}$-only execution that exists by the boundary condition and let $C' = C\alpha$. If there is a $\mathcal{P}$-only execution α' from C' such that $\mathcal{P}$ has $k+1$ valencies in $C'\alpha'\beta$, then $\gamma = \alpha\alpha'$ satisfies the claim. So, suppose that, or every $\mathcal{P}$-only execution α' from C', $\mathcal{P}$ has at most k valencies in $C'\alpha'\beta$. From the boundary condition, for each $0 \leq i \leq k$, there exists $p_i \in \mathcal{P}$ such that p_i has valency i in C'. We now construct a wait-free protocol for k-set agreement task among $k+1$ processes $\mathcal{Q} = \{q_0, \dots q_k\}$, again contradicting Theorem 3.4.1. The idea is to have processes in $\mathcal{Q}$ simulate processes in $\{p_0, \dots, p_k\}$, and also a block write by $\mathcal{R}$, but only after every q_i has taken a step.

The simulation uses a multi-writer snapshot object with one component for each register in the original protocol, initialized to its contents in C', as well as $k+1$ additional one bit components, $b_0, b_1, \dots, b_k$, each initialized to 0. These bits indicate which processes in $\mathcal{Q}$ have entered the simulation: Each process q_i updates b_i to 1 with its first step. Then, as in Lemma 3.4.2, q_i immediately returns i if p_i has already decided i in C'. The rest of the simulation works as follows.

Process q_i simulates p_i, updating the components corresponding to which p_i writes, performing scans to simulate p_i's reads, and returning value v if p_i decides v in the simulation. If q_i ever performs a scan and sees that every process in $\mathcal{Q}$ has taken a step, i.e. $b_0 = b_1 = \cdots = b_k = 1$, then q_i considers the contents of the components corresponding to the registers in the view returned by this latest scan. It locally emulates the pending block write β by $\mathcal{R}$ (changing the emulated contents of $|\mathcal{R}|$ registers) followed by a solo execution of p_i, which terminates by obstruction-freedom. Note that q_i does not take any more steps after this scan and decides whatever p_i decides in the local emulation. To see that this simulation is correct, we distinguish two types of executions:

Case 1: *Some process in $\mathcal{Q}$ does not take steps.* At most k simulated processes take steps and wait-freedom is guaranteed by k-obstruction-freedom. Furthermore, at most k values are returned. By the boundary condition in C', if value v is decided by some q_i, then p_v and, thus, q_v have taken at least one step.

Case 2: *Every process in $\mathcal{Q}$ takes at least one step.* Consider the first configuration D of the simulating system where every process in $\mathcal{Q}$ has taken at least one step. Then, the simulated system is in some configuration $C'\alpha'$. We show that any valency v of $\mathcal{Q}$ in D in the simulating system is also a valency of $\mathcal{P}$ in $C'\alpha'\beta$ in the simulated system. Since $\mathcal{P}$ has at most k valencies in $C'\alpha'\beta$, processes in $\mathcal{Q}$ will never violate k-set agreement. Moreover, any process q_i that has not already decided in D will return after performing its next scan. Thus the simulation is wait-free.

For any process q_i that has already decided v in D or will decide v after next performing an update, the value v must be the valency of p_i in $C\alpha'$ and thus, a valency of $p_i \in \mathcal{P}$ in $C'\alpha'\beta$, as desired. Now consider the remaining processes $\mathcal{Q}' \in \mathcal{Q}$. Each is about to perform a scan or an update operation followed by a scan on behalf of the process it is simulating. In its scan it will observes that every process in $\mathcal{Q}$ has taken a step and then do local computation to decide a value. Therefore, each $q_i \in \mathcal{Q}'$ does its emulation of p_i from a configuration $C\alpha'\mu_i\beta$, where μ_i consists of some of the pending writes of processes p_j with $q_j \in \mathcal{Q}'$. In $C\alpha'\mu_i\beta$, the contents of registers are the same as in configuration $C\alpha'\beta\mu_i'$, where μ_i' contains the writes from μ_i to the registers not written to by β (i.e. not covered by $\mathcal{R}$), in the same order as in

μ_i. Suppose q_i decides v, which means that p_i decides v from $C\alpha'\beta\mu_i'$ in the emulated solo execution. Since μ_i' consists of steps by processes in $\mathcal{P}$, v must be a valency of $\mathcal{P}$ in $C'\alpha'\beta$, completing the proof. $\square$

Lemma 3.4.4. *Any k-set agreement protocol that satisfies obstruction-freedom has an initial configuration from which a set of $k+1$ processes $\mathcal{P}$ has valencies $\{0, 1, \ldots, k\}$.*

Proof. For each $v \in \{0, 1, \ldots, k\}$, let I_v be the configuration where every process starts with input v. Now consider an initial configuration I, where, for each $v \in \{0, 1, \ldots, k\}$, there is a process $p_v \in \mathcal{P}$ that starts with input v. Configurations I and I_v are indistinguishable to p_v, so it returns v in a solo execution starting from I. Therefore, $\mathcal{P}$ has $k+1$ valencies in configuration I. $\square$

We use Lemma 3.4.3 to perform block writes that allow covering more registers, maintaining all $k+1$ valencies by Lemma 3.4.2. We get the following bound.

Theorem 3.4.5. *Any k-obstruction-free protocol for solving k-set agreement among n processes that satisfies the boundary condition uses at least $n-k-1$ registers.*

Proof. We start in the initial configuration I provided by Lemma 3.4.4, with a set of processes $\mathcal{P}$ that have valencies $\{0, 1, \ldots, k\}$ in I. The set $\mathcal{P}$ will not change during the argument. Let $\{p_1', \ldots, p_{n-k-1}'\}$ be the set of remaining processes and, for $0 \leq r \leq n-k-1$, let $\mathcal{R}_r = \{p_j' \mid 1 \leq j \leq r\}$. We will prove by induction on r, that from any configuration where $\mathcal{P}$ has $k+1$ valencies, it is possible to reach a configuration by $(\mathcal{P} \cup \mathcal{R}_r)$-only execution, in which processes in $\mathcal{R}_r$ cover r different registers and processes in $\mathcal{P}$ still have $k+1$ different valencies. This proves the existence of a configuration with $n-k-1$ covered registers.

The base case $r = 0$ of the induction holds in any configuration where $\mathcal{P}$ has $k+1$ valencies.

Now suppose the induction hypothesis holds for $r < n-k-1$ and consider any reachable configuration C in which $\mathcal{P}$ has $k+1$ valencies (eg. the initial configuration). We construct a sequence of configurations $C_0, C_1, \ldots$ reachable from C such that, for all $i \geq 0$, the following properties hold:

1. In C_i, the processes in $\mathcal{R}_r$ cover r different registers,
2. In C_i, $\mathcal{P}$ has valencies $\{0, 1, \ldots, k\}$, and
3. C_{i+1} is reachable from C_i by a $(\mathcal{P} \cup \mathcal{R}_r)$-only execution α_i which contains a block write β_i by $\mathcal{R}_r$ to the r registers covered by $\mathcal{R}_r$ in C_i.

By the induction hypothesis, there is a $(\mathcal{P} \cup \mathcal{R}_r)$-only execution η such that, in configuration $C_0 = C\eta$, $\mathcal{R}_r$ covers r different registers and $\mathcal{P}$ has $k+1$ valencies. If $r = 0$, let $C_i = C_0$ and α_i be an empty execution, for all $i \geq 1$. Otherwise, let $i \geq 0$ be arbitrary and suppose we have constructed C_i with the desired properties. By Lemma 3.4.3, there is a $\mathcal{P}$-only execution γ_i from C_i such that $\mathcal{P}$ has $k+1$ valencies in $C_i\gamma_i\beta_i$, where β_i is the block write by $\mathcal{R}_r$ to the r registers it covers in C_i. By the induction hypothesis, there is a $(\mathcal{P} \cup \mathcal{R}_r)$-only execution η_i such that, in configuration $C_{i+1} = C_i\gamma_i\beta_i\eta$, $\mathcal{R}_r$ covers r different registers and $\mathcal{P}$ still has valencies $\{0, 1, \ldots, k\}$. Let $\alpha_i = \gamma_i\beta_i\eta_i$. Then C_{i+1} satisfies the three desired properties.

There are only finitely many registers, so there exists $0 \leq i < j$ such that $\mathcal{R}_r$ covers the same set of registers in C_i as in C_j. We now insert steps of process p'_{r+1} so that no process in $\mathcal{P} \cup \mathcal{R}_r$ can detect them. Consider any $\{p'_{r+1}\}$-only execution ζ from $C_i\gamma_i$ that decides a value. If ζ only writes to registers covered by $\mathcal{R}_r$ in C_i, then $C_i\gamma_i\zeta\beta_i$ is indistinguishable from $C_i\gamma_i\beta_i$ to processes in $\mathcal{P}$, due to the block write β_i. Hence $\mathcal{P}$ has $k+1$ valencies in configuration $C_i\gamma_i\zeta\beta_i$. But this contradicts Lemma 3.4.2, since process p'_{r+1} has decided in configuration $C_i\gamma_i\zeta\beta_i$. Therefore, during ζ, process p'_{r+1} writes to a register not covered by $\mathcal{R}_r$ in C_i. Let ζ' be the longest prefix of ζ containing only writes to registers covered by $\mathcal{R}_r$ in C_i. Then, in $C_i\gamma_i\zeta'$, p'_{r+1} is poised to perform a write to a register not covered by $\mathcal{R}_r$ in C_i (or in C_j).

$C_i\gamma_i\zeta'\beta_i$ is indistinguishable from $C_i\gamma_i\beta_i$ to $\mathcal{P} \cup \mathcal{R}_r$, so the $\mathcal{P} \cup \mathcal{R}_r$-only execution $\eta_i\alpha_{i+1} \cdots \alpha_{j-1}$ is still applicable from $C_i\gamma_i\zeta'\beta_i$. Let $\alpha = \eta\alpha_0 \cdots \alpha_{i-1}\gamma_i\zeta'\beta_i\eta_i\alpha_{i+1} \cdots \alpha_{j-1}$. Every process in $\mathcal{P} \cup \mathcal{R}_r$ is in the same state in $C\alpha$ as in C_j. In particular, $\mathcal{P}$ has $k+1$ valencies and $\mathcal{R}_r$ covers the same r registers in $C\alpha$ as in C_j. Moreover, since p'_{r+1} takes no steps in $\beta_i\eta_i\alpha_{i+1} \cdots \alpha_{j-1}$, it covers a location in $C\alpha$ that is not covered by $\mathcal{R}_r$ in C_i (or C_j). Therefore, in $C\alpha$, the set $\mathcal{R}_{r+1} = \mathcal{R} \cup \{p'_{r+1}\}$ covers a set of

$r+1$ registers, as desired. $\square$

3.4.3 Global Argument

Consider any k-set agreement protocol Π for $n > k$ processes, $p_1, \ldots, p_n$, that satisfies k-obstruction-freedom. We will prove that the protocol must use at least $n-k+1$ registers. Indeed, *assume that the protocol uses $m \leq n-k$ registers.* We show that, in this case, it is possible to simulate Π and deterministically solve k-set agreement among $k+1$ processes, say $q_0, q_1, \ldots, q_k$, in a wait-free manner. Since this is known to be impossible (see Theorem 3.4.1), the result follows.

Overview: We start by an intuitive overview of the simulation. Suppose q_i starts with input i. Processes $q_1, \ldots, q_k$ directly simulate $p_{n-k+1}, \ldots, p_n$ on their respective inputs and return what the simulated process decides if it terminates in the simulation. We pay special attention to process q_0, as it will simulate processes $p_1, \ldots, p_m$, all with input 0. If any of these processes returns in the simulation, q_0 returns the same value. Then, since Π is k-obstruction-free, $p_{n-k+1}, \ldots, p_n$, and hence $q_1, \ldots, q_k$, will all eventually decide. q_0 starts by simulating p_1. If p_1 does not perform any more writes after some point, then $p_{n-k+1}, \ldots, p_n$ cannot tell if p_1 actually takes any more steps, and they must eventually all decide. Hence, so will p_1. So suppose that p_1 always performs more writes and does not terminate.

Eventually, q_0 will be about to simulate a write by p_1 to a register r_1 that p_1 has previously written to. *Assuming q_0 can determine what the contents of the registers were in Π immediately before p_1's first write to r_1*, q_0 now *locally* simulates p_2 running solo immediately before p_1's previous write to r_1.

If p_2 does not write to any registers other than r_1 and decides some value, then q_0 decides the same value and terminates. This is valid because p_2 could actually have taken these steps before p_1's first write to r_1 (which hides them) and the other processes $p_{n-k+1}, \ldots, p_n$ would not be able to tell the difference. Thus, as before, the other processes are guaranteed to all decide afterwards.

So suppose p_2 writes to a register $r_2 \neq r_1$ when it runs solo immediately before

p_1's first write to r_1. In this case, q_0 locally simulates p_2, until p_2 is about to write to r_2. Now, instead of just simulating p_1's write to r_1, q_0 *signals* to the other simulators that $\{p_1, p_2\}$ performed a block write to $\{r_1, r_2\}$. After this, q_0 continues simulating p_1 (and p_2 as appropriate, maintaining its state locally) as described.

Continuing to simulate processes $p_1, p_2, \ldots, p_{n-k}$ this way, q_0 either decides, or signals larger and larger block writes. Eventually, q_k will be about to signal a block write by $\{p_1, p_2, \ldots, p_m\}$ to $m \leq n-k$ registers (or it returns). We assumed that Π uses m registers so this block write obliterates the contents of all registers. Thus q_0 can locally simulate what p_1 decides after the block update by $\{p_1, \ldots, p_m\}$ and decide that value. After this, all other processes will also decide by k-obstruction-freedom. Hence, the simulation is wait-free, which is impossible.

An astute reader may have observed that q_0 simulates processes in a manner reminiscent of the adversarial scheduler in the $n-1$ lower bound for consensus. Indeed, viewing the adversarial scheduler as an algorithm, q_0 is essentially running that algorithm. In some sense, this pits the space lower bound adversary against the k-set agreement impossibility adversary. This is a win-win situation for us. To avoid agreement due to the k-obstruction-freedom, q_0, and thus, the space lower bound adversary, must take steps. But the more steps the space lower bound adversary takes, the more registers get covered.

Critical to the simulation is the ability to emulate a system with m multi-writer registers, such that one process, q_0, can also signal block writes. In addition, q_0 needs to determine the contents of registers at certain points before the block writes, in order to locally simulate the extra process for a subsequent larger block in a consistent way. This is accomplished by the following object implementation from read-write registers.

1-Augmented Snapshot Object: We implement a 1-*augmented* m-*component multi-writer snapshot object* $\mathcal{M}[1..m]$ shared by $k+1$ processes, $q_0, q_1, \ldots, q_k$. It supports an *Update*(j, v) operation, for $1 \leq j \leq m$, that can be performed by processes q_i, for $1 \leq i \leq k$, and a *Scan*() operation that can be performed by all processes.

In addition, it supports a *Block-Update*$([j_1, \ldots, j_\ell], [v_1, \ldots, v_\ell])$ operation that can be performed only by process q_0.

Update(j, v) updates the value of the j-th component of $\mathcal{M}$ to v and *Scan*() returns the values of all components of $\mathcal{M}$. A *Block-Update* B to components $[j_1, \ldots, j_\ell]$ of $\mathcal{M}$ sets $\mathcal{M}[j_r]$ to v_r for all $1 \leq r \leq \ell$, and returns a *Scan* of $\mathcal{M}$ with the following properties: Let L be the last *Block-Update* by q_0 prior to B, or the beginning of the execution, if B is the first *Block-Update* by q_0. Let L' be the last *Scan* operation between L and B, or L, if there is no such operation. Then the *Block-Update* returns the values of all components of $\mathcal{M}$ at some point between L' and B. This is the key property that allows q_0 to insert steps of new processes into the simulation.

We design a non-blocking linearizable [HW90] implementation of $\mathcal{M}$. It uses a single-writer snapshot object $\mathsf{H}[0..k]$, along with an unbounded number of single-writer registers $\mathsf{L}_1[b], \ldots, \mathsf{L}_k[b]$ for $b \geq 0$, that are only written to by processes $q_1, \ldots, q_k$, respectively. As customary, H supports atomic *update*(j, v) and *scan*() methods, while $\mathsf{L}_i[b]$ supports atomic *read*() and *write*(v) methods. Component $\mathsf{H}[i]$ is used by q_i to store the history of its *Updates* as a list of triples (j, v, t), where j is a component of $\mathcal{M}$, v is the value, and t is a unique *timestamp* associated with the update. For a *Block-Update* of ℓ components, q_0 appends ℓ triples to $\mathsf{H}[0]$, all with the same timestamp. Registers $\mathsf{L}_i[b]$ are used to help q_0 determine what to return in its $b + 1$'st *Block-Update*. Initially, $\mathsf{H} = (\bot, \ldots, \bot)$ and $\mathsf{L}_i[b] = \bot$ for all $1 \leq i \leq k$ and all $b \geq 0$.

Notation: We will use uppercase latin letters to denote execution steps (*scan*, *update*, *read*, *write* invocations) and operations (*Update*, *Scan*, *Block-Update*). The corresponding lowercase letters denote return values. For instance, if H is a *scan* (H.*scan*()), then h will be the *result of scan* H (return value of H). Given h, a result of *scan*, we denote by h_i the value in i-th component (of $H[i]$), and by $\#h_i$ the number of different timestamps associated with the triples recorded in h_i. For process $q_i \neq q_0$, this will be the number of *Update* operations it has performed on $\mathcal{M}$. For process q_0, $\#h_0$ will be the number of *Block-Update*s.

The pseudo-code of our implementation of $\mathcal{M}$ is given in Figure 3-5.

```
procedure $Get\text{-}timestamp(h)$ by $q_i$
    for $j \in \{0, 1, \ldots, k\}$ do
        $t_j \leftarrow \#h_j$
    $t_i \leftarrow t_i + 1$
    return $(t_0, \ldots, t_k)$
procedure $Get\text{-}view(h)$ by $q_i$
    for $j = 1..m$ do
        if there is an update triple in $h$ with first component $j$ then
            $t \leftarrow \max\{t' : (j, v', t')$ is a triple in $h\}$
            let $(j, v, t)$ be the *unique* triple in $h$ with component $j$ and timestamp $t$
            $v_j \leftarrow v$
        else $v_j \leftarrow \perp$
    return $(v_1, \ldots, v_m)$
procedure $Scan()$ by $q_i$
    repeat
        $h \leftarrow \mathsf{H}.scan()$
        if $i \neq 0$ then $\mathsf{L}_i[\#h_0].write(h)$
        $h' \leftarrow \mathsf{H}.scan()$
    until $h = h'$
    return $Get\text{-}view(h)$
procedure $Update(j, v)$ by $q_i \neq q_0$
    $h \leftarrow \mathsf{H}.scan()$
    $t \leftarrow Get\text{-}timestamp(h)$ // Get a timestamp, $t$, associated with the *Update*
    $\mathsf{H}.update(i, h_i \cdot (j, v, t))$        // Append $(j, v, t)$ to the $i$'th component of H
procedure $Block\text{-}Update([j_1, \ldots, j_\ell], [v_1, \ldots, v_\ell])$ by $q_0$
    $h \leftarrow \mathsf{H}.scan()$
    $t \leftarrow Get\text{-}timestamp(h)$
    $\mathsf{H}.update(0, h_0 \cdot [(j_1, v_1, t), \ldots, (j_\ell, v_\ell, t)])$            // Append all to the 0'th
    component of H
    $\mathsf{last} \leftarrow h$
    for $j \in \{1, \ldots, k\}$ do
        $r[j] \leftarrow \mathsf{L}_j[\#h_0].read()$
        if $r[j] \neq \perp$ and $\mathsf{last}$ is a proper prefix of $r[j]$ then
            $\mathsf{last} \leftarrow r[j]$
    return $Get\text{-}view(\mathsf{last})$
```

Figure 3-5: Pseudocode for the Implementation of a 1-Augmented Snapshot Object

Auxiliary Procedures: As in [Mat89, Fid91, AW04], a *timestamp* is a $(k+1)$-component vector of non-negative integers, ordered lexicographically. Given h, a result of *scan*, process q_i *generates a new timestamp* $t = (t_0, t_1, \ldots, t_k)$ from h by setting $t_j \leftarrow \#h_j$ for $j \neq i$ and $t_i \leftarrow \#h_i + 1$. In Lemma 3.4.8 we show that timestamps are unique.

Given h, a result of *scan*, for each component j, let v_j be the value with the lexicographically largest associated timestamp among all triples (j, v, t) in all components of h, or $\perp$ if no such triple exists. The *view* of h, denoted *view*(h), is the vector $(v_1, \ldots, v_m)$.

Main Procedures: To perform an *Update*(j, v), process $q_i \neq q_0$ (q_0 always uses *Block-update*) *scan*s H to obtain h, computes the timestamp t from h that will be associated with the *Update*, and *update*s $\mathsf{H}[i]$ by appending (j, v, t) to it.

To compute a return value of a *Block-Update*, q_0 uses the *latest* scan of H taken by any process before the *Block-Update*. To make this possible, each process $q_i \neq q_0$ helps q_0 while executing the *Scan* operation. During *Scan*(), q_i repeatedly *scan*s H. It records the result h of its *scan* in $\mathsf{L}_i[\#h_0]$ (recall that $\#h_0$ represents the number of *Block-Update*s by q_0). Then it takes a second *scan* of H and retries helping if the result of the *scan* is no longer equal to h. Otherwise, it returns *view*(h).

To perform a *Block-update*$([j_1, \ldots, j_\ell], [v_1, \ldots, v_\ell])$, q_0 first takes a scan H of H. It then generates a timestamp t from h (the result of H) to associate with the *Block-Update* and *update*s $\mathsf{H}[0]$ by appending the triples $(j_1, v_1, t), \ldots, (j_\ell, v_\ell, t)$ to it. To determine the return value, q_0 takes another *scan* of H, and sets last to the result of the latest *scan* among the result of this *scan* and $\{\mathsf{L}_i[b] : \mathsf{L}_i[b] \neq \perp\}$, where b is the number of *Block-Update*s by q_0. Notice that any two *scan*s H and H' are comparable component-wise, so q_0 can detect if a *scan* step occured earlier than another *scan* step by checking if some component of the result has less entries (formally, see Observation 2). Finally, q_0 removes all triples with timestamp t from f_0, i.e. the records of the ongoing *Block-Update*, and returns the view.

Linearization Points: A *Scan* operation is linearized at the last *scan* of H in performed in the *Scan* on Line 70. A (*Block-*)*Update* operation with associated timestamp t involving component j is linearized at the first point that H contains an update triple with component j and timestamp $t' \succeq t$. If multiple (*Block-*)*Update* operations are linearized at the same point, then they are ordered in increasing order of their associated timestamps. The following observations are consequences of these definitions.

Observation 1. *Let U be an Update operation to component j with an associated timestamp t and let X be any update to H that appends an update triple with component j and timestamp $t' \succeq t$ to H. Then U is linearized no later than X. A Block-Update operation B is always linearized at its update to H. Furthermore, if multiple Updates are linearized at B's update to H, then B is ordered last.*

If H and H' are scans of H with results h and h' such that, for each $j \in \{0, 1, \ldots, k\}$, h_j is a prefix of h'_j, then we say that h is a *prefix* of h'. If h_j is a proper prefix of h'_j for some j, then we say that h is a *proper* prefix of h'. Since processes only append values to their own components in H and *scans* of H are atomic, we have the following observation.

Observation 2. *Let H and H' be* scans *of H that return h and h', respectively. If H occurs before H', then, for each $j \in \{0, 1 \ldots, k\}$, h_j is a prefix of h'_j. Conversely, if there is some $j \in \{0, 1 \ldots, k\}$ such that h_j is a proper prefix of h'_j, then H occurs before H'.*

We say that the result h of a *scan* of H *contains* a timestamp t, if h (or, more precisely, some component h_i of h) contains an update triple with timestamp t. The next lemma implies that a timestamp generated from h by any process is lexicographically larger than any timestamp contained in h.

Lemma 3.4.6. *For any timestamp t contained in the result h' of a scan H' of H, $\#h'_j \geq t_j$, for all $0 \leq j \leq k$.*

Proof. By definition, t is generated from a result h of a scan H by some process q_i as follows: $t_j = \#h_j$, for $j \neq i$, and $t_i = \#h_i + 1$. Since H occurs before H', by Observation 2, $\#h'_j \geq \#h_j = t_j$, for all $j \neq i$. Furthermore, h' contains t, so q_i's *update* to H that appends an update triple with timestamp t occurs before H', implying that $\#h'_i \geq \#h_i + 1 = t_i$. $\square$

Corollary 3.4.7. *Let h' be the result of a scan and let $t' =$ Get-timestamp(h') by any process. Then, for any timestamp t contained in h', $t \prec t'$.*

Lemma 3.4.8. *Every (Block-)Update operation is associated with a different timestamp.*

Proof. Suppose two processes $q_i \neq q_j$ generate timestamps t and t' from *scans* H and H' of H that return h and h', respectively. Then $t_i = \#h_i + 1$, $t_j = \#h_j$, $t'_j = \#h'_j + 1$, and $t'_i = \#h'_i$. If $t = t'$, then $\#h_i + 1 = \#h'_i$ and $\#h'_j + 1 = \#h_j$. It follows that $\#h_i < \#h'_i$ and $\#h_j > \#h'_j$. However, by Observation 2, this is impossible. Therefore, $t \neq t'$.

Now, consider timestamps generated by the same process q_i. Since q_i appends one or more updates triples with timestamp t to the i-th component of H immediately after it generates t, the result of any subsequent scan by q_i contains t. Thus, by Corollary 3.4.7, any timestamp t' generated by q_i after t is lexicograpically larger than t. □

Now, we prove that our linearization is correct. We first show that the linearization points of operations are always contained within their respective execution intervals.

Lemma 3.4.9 (Linearization)**.** *Each operation is linearized within its execution interval.*

Proof. By definition, a *Scan* operation is linearized at a point within its execution interval. A (*Block-*)*Update* operation U to component j with associated timestamp t is linearized at the first point that H contains an update triple with component j and timestamp $t' \succeq t$. Let h be the result of the *scan* H in U. Then $t =$ *Get-timestamp*(h). By Corollary 3.4.7, all of the timestamps contained in h are lexicographically smaller than t. Thus, U is linearized after H. On the other hand, since U contains an *update* to H that appends an update triple with component j and timestamp t, U is linearized no later than this *update*. It follows that U is linearized at a point in its execution interval. □

Observation 3. *If a scan H' of* H *occurs after the linearization point of a (Block-) Update U involving component j with associated timestamp t, then the result of H' contains an update triple with component j and timestamp at least as large as t.*

The next lemma says that if H is a *scan* that returns h, then *Get-view*(h) is the result that should be returned by any *Scan* linearized at H according to the specification of $\mathcal{M}$.

Lemma 3.4.10. *Let H be a scan that returns h. Suppose Get-view$(h) = (v_1, \ldots, v_m)$. Then, for each $1 \leq j \leq m$, v_j is the value of the last (Block-) Update to component j of $\mathcal{M}$ linearized before H, or $\perp$ if no such (Block-) Update exists.*

Proof. First, suppose that no (*Block-*) *Update* operation involving component j is linearized before H. We claim that h does not contain an update triple involving component j. Assume the contrary. Among all update triples with component j in h, let (j, v_j, t) be the one with the largest timestamp. Some (*Block-*) *Update* operation U performed an *update* X that appended (j, v_j, t) to H prior to H. However, by Observation 1, U would be linearized no later than X. This is a contradiction, since X occurs before H and, hence, U is linearized before H.

Now, suppose there is some (*Block-*) *Update* operation involving component j that is linearized before H. Consider the last such operation U linearized before H. Let t be its associated timestamp. Then, by Observation 3, h contains an update triple with component j and timestamp $t'' \succeq t$. Let t' be the largest timestamp of any update triple with component j in h. By Lemma 3.4.8, there is exactly one update triple in h with component j and timestamp t'. By definition of *Get-view*(h), the value of this update triple is v_j.

Since h contains an update triple with component j and timestamp t'', we have that $t' \succeq t'' \succeq t$. We now show that $t' = t$ which, by Lemma 3.4.8, implies that U appended triple (j, v_j, t'), as desired. Indeed, let X be the *update* to H that appended (j, v_j, t') and let U' be the (*Block-*) *Update* operation that performed X. By definition of t', prior to X, H does not contain an update triple with component j and timestamp at least t'. By our linearization rules, it follows that U' is linearized at X, which is before H.

On the other hand, by Observation 1, since $t' \succeq t$, U is linearized at no later than X. Since U is the last (*Block-*)*Update* operation involving component j linearized before H and U' is linearized at X, U must be linearized at X. If $t' \succ t$, then, by our linearization rules, U is linearized earlier than U'. This contradicts the definition of U. Therefore, $t = t'$. □

Corollary 3.4.11 (*Scans*). *Consider any Scan that returns* $(v_1, \ldots, v_m)$. *Then, for each* $1 \leq j \leq m$, v_j *is the value of the last* (*Block-*)*Update to component* j *of* $\mathcal{M}$ *linearized before the Scan operation, or* $\perp$ *if no such* (*Block-*)*Update exists.*

Lemma 3.4.12. *Let* U *be a Block-Update operation, let* H *be the scan of* H *in* U *that returns* h, *let* X *be the update to* H *in* U, *let* $r[1], \ldots, r[k]$ *be the values read from* $\mathsf{L}_1[\#h_0], \ldots, \mathsf{L}_k[\#h_0]$, *respectively, in* U, *and let* last *be the value of the variable on Line 81 after the* **for** *loop in* U. *Then:*

- last *is the result of a scan that occurs before* X.
- *Let* L *be the last scan that returns* last. *Then* L *occurs no earlier than* H *and there is no Scan operation linearized between* L *and* X.

Proof. A process $q_i \neq q_0$ only writes to $\mathsf{L}_i[b]$ when it takes a *scan* H' of H that returns h' such that $\#h'_0 = b$. Thus, at all times, $\mathsf{L}_i[\#h_0]$ contains either $\perp$ or the result of a scan H' of H taken before X. It follows that $r[1], \ldots, r[k]$ are all results of scans of H that occur before X. Therefore, last, which is either h or some $r[j]$, can only be the the result of a *scan* of H that occurs before X. By the code, h is a prefix of last. Since L is the last *scan* to return last, by Observation 2, L occurs no earlier than H.

Now suppose for a contradiction, that there is some *Scan* operation S linearized between L and X. Let G be the last *scan* in S, where S is linearized, and let g be the result of G. Since G is between L and X, hence, between H and X, S is not performed by q_0. Suppose S is performed by $q_i \neq q_0$.

Since G occurs after L, which occurs no earlier than H, by Observation 2, $\#g_0 \geq \#h_0$. On the other hand, since G occurs before X, $\#g_0 \leq \#h_0$. It follows that

$\#g_0 = \#h_0$. From the condition on Line 71, this implies that q_i performed a *write* to $\mathsf{L}_i[\#h_0]$ with value g before G. Since G occurs before X and $r[i]$ is read from $\mathsf{L}_i[\#h_0]$ after X, g is a prefix of $r[i]$. By definition of last, it follows that g is a prefix of last. On the other hand, since L occurs before G, last is a prefix of g. Therefore, $\mathsf{last} = g$. It follows that G occurs no later than L. Contradiction. $\square$

Corollary 3.4.13 (*Block-Update*s)**.** *Let B be a Block-Update operation by q_0 and let X be the update of* H *at which B is linearized. Consider the Block-Update by q_0 prior to B, and let X' be its linearization point, or the beginning of the execution if B is the first Block-Update. If there is a Scan operation linearized between X' and X, then let S be the linearization point of the last Scan operation linearized between X' and X. Otherwise, let S be X'. Then B returns the values of components of $\mathcal{M}$ linearized at some point between S and X.*

Proof. Let last and L be defined as in Lemma 3.4.12. By Lemma 3.4.12, L occurs no earlier than the scan of H in B, and hence L occurs after X', since X' is either the beginning of the execution or an *update* performed by q_0 earlier. Furthermore, L occurs before X. It follows that L occurs after S, since otherwise, there would be a *Scan* linearized between L and X, contradicting Lemma 3.4.12. Since L returns last, by Lemma 3.4.10, *Get-view*(last) returns the values of the last (*Block-*)*Update*s to each component linearized before L, as desired. $\square$

Theorem 3.4.14. *The algorithm in Figure 3-5 is a linearizable non-blocking implementation of a* 1*-augmented m-component multi-writer snapshot object $\mathcal{M}$.*

Proof. Linearizability follows from Lemma 3.4.9, Corollary 3.4.11 and Corollary 3.4.13. By the code, (*Block-*)*Update* operations are wait-free. On the other hand, if a process takes steps but does not return from a *Scan* invocation, then it must repeatedly be failing the condition on Line 71. This is only possible if a new triple is appended to H by an *update* on Line 76 or Line 80 in an unique corresponding (*Block-*)*Update* operation. Thus, other processes must be completing infinitely many (*Block-*)*Update* invocations. $\square$

Theorem 3.4.15. *Any k-obstruction-free protocol using read-write registers for k-set agreement among $n > k$ requires $n - k + 1$ registers.*

Proof. Suppose not, so that the protocol uses $m \leq n - k$ registers. Without loss of generality, we assume that the protocol uses an m-component multi-writer snapshot. We will show that it is possible for $k + 1$ processes $q_0, q_1, \ldots, q_k$ to solve wait-free k-set agreement by simulating the protocol using our 1-augmented snapshot object $\mathcal{M}[1..m]$, contradicting Theorem 3.4.1.

Indeed, $q_1, \ldots, q_k$ directly simulate steps of $p_{n-k+1}, \ldots, p_n$ on their respective inputs using $\mathcal{M}$ to perform updates and snapshots. Process q_0 simulates $p_1, \ldots, p_m$ on its input according to the space lower bound adversary, by the strategy described in the beginning of Section 3.4.3. Initially, it simulates p_1, using *Block-Update* operation to *build* block updates of size 1, and decides if p_1 decides.

To *build* block update of size $1 < s \leq m$, q_0 repeatedly builds block updates of size $s - 1$ (using $p_1, \ldots, p_{s-1}$), keeping track of the views returned by the *Block-Update*s as well as the states of the simulated processes, until it is about to perform a block update to a set of components C of size $s - 1$ that it has previously performed a *Block-Update* U to. It then locally simulates p_s running solo from the view returned by U. If p_s decides a value before updating a component not in C, then q_0 returns that value. Otherwise, it stops p_s in its local simulation before p_s is about to perform an update to a component $c \notin C$. Finally, if $s < m$, then it uses $\mathcal{M}$ to signal a block update of size s to components $C \cup \{c\}$, including p_s's update in addition to the original block update of size $s - 1$. If $s = m$, then q_0 locally simulates p_1 running solo after the block update by $p_1, \ldots, p_s$ to components $C \cup \{c\}$ and decides the same value as p_1 (and terminates).

We claim that this simulates a valid execution of the protocol. Indeed, we linearize steps by the processes in order of their operations on $\mathcal{M}$. Block updates by processes in $\{p_1, \ldots, p_m\}$ signaled by q_0 are linearized consecutively, as a block, as desired. Furthermore, the locally simulated steps of processes $p_2, \ldots, p_m$ are inserted immediately after the corresponding views returned by *Block-Update* from which their solo executions are simulated from. This is indistinguishable to every other simulated process because the steps are hidden by the block updates and because, by the property on the views that *Block-Update*s return, no process takes a scan between their inserted steps and before the *Block-Update*.

Finally, the simulation is wait-free. Indeed, in any execution where q_0 takes finitely many steps, the remaining processes $q_1, \dots, q_k$ are guaranteed to terminate because the protocol is k-obstruction-free. Furthermore, after q_0 performs at most

$$\left(\binom{m}{1}+1\right)\left(\binom{m}{2}+1\right)\dots\left(\binom{m}{m-1}+1\right) \leq \left(2\binom{m}{m/2}\right)^{m-1} \leq 2^{m^2}$$

block updates, it is guaranteed to terminate as it will have built a block update to m components. After this, $q_1, \dots, q_k$ are guaranteed to terminate as well because of k-obstruction-freedom. Contradiction. □

Chapter 4

Message Passing

4.1 Definitions and Notation

We consider the classic asynchronous message-passing model [ABND95]. Here, n processors communicate with each other by sending *messages* through *channels*. There is one channel from each processor to every other processor; the channel from i to j is independent from the channel from j to i. Messages can be arbitrarily delayed by a channel, but do not get corrupted.

Computations are modeled as sequences of steps of the processors, which can be either *delivery steps*, representing the delivery of a new message, or *computation steps*. At each computation step, the processor receives all messages delivered to it since the last computation step, and, unless it is *faulty*, it can perform local computation and send new messages. A processor is *non-faulty*, if it is allowed to perform local computations and send messages infinitely often and if all messages it sends are eventually delivered. Notice that messages are also delivered to *faulty* processors, although their outgoing messages may be dropped.

We consider algorithms that tolerate up to $t \leq \lceil n/2 \rceil - 1$ processor failures. That is, when more than half of the processors are non-faulty, they all return an answer from the protocol with probability one. A standard assumption in this setting is that all non-faulty processors always take part in the computation by replying to the messages, irrespective of whether they participate in a certain algorithm or even after they return a value—otherwise, the $t \leq \lceil n/2 \rceil - 1$ condition may be violated.

Communicate Procedure: Our algorithms use a procedure called communicate, defined in [ABND95] as a building block for asynchronous communication. The call communicate(m) sends the message m to all n processors and waits for at least $\lfloor n/2 \rfloor + 1$ acknowledgments before proceeding with the protocol. The communicate procedure can be viewed as a best-effort broadcast mechanism; its key property is that any two communicate calls intersect in at least one recipient. In the following, a processor i will communicate messages of the form ($propagate$,v_i) or ($collect$,v). For the first message type, each recipient j updates its view of the variable v and acknowledges by sending back an ACK message. In the second case, the acknowledgement is a pair (ACK,v_j) containing j's view of the variable for the receiving process. In both cases, processor i waits for $> n/2$ ACK replies before proceeding with its protocol. In the case of *collect*, the communicate call returns an array of at least $\lfloor n/2 \rfloor + 1$ views that were received.

Adversary: We consider strong adversarial setting where the scheduling of processor steps, message deliveries and processor failures are controlled by an adaptive adversary. At any point, the adversary can examine the system state, including the outcomes of random coin flips, and adjusts the scheduling accordingly.

Complexity Measures: We consider two worst-case complexity measures against the adaptive adversary. *Message complexity* is the maximum expected number of messages sent by all processors during an execution. When defining *time complexity*, we need to take into account the fact that, in asynchronous message-passing, the adversary schedules both message delivery and local computation.

Definition 4.1.1 (Time Complexity)**.** *Assume that the adversary fixes two arbitrarily large numbers t_1 and t_2 before an execution, and these numbers are unknown to the algorithm. Then, during the execution, the adversary delivers every message of a non-faulty processor within time t_1 and schedules a subsequent step of any non-faulty* *processor in time at most t_2.*[1] *An algorithm has time complexity $O(T)$ if the maximum expected time before all non-faulty processors return that the adversary can achieve is $O(T(t_1 + t_2))$.*[2]

For instance, in our algorithms, all messages are triggered by communicate. A processor depends on the adversary to schedule a step in order to compute and call communicate, and then depends on the adversary to deliver these messages and acknowledgments. In the above definition, if all processors call communicate at most T times, then all non-faulty processors return in time at most $2T(t_1+t_2) = O(T(t_1+t_2))$: each communicated message reaches destination in time t_1, gets processed within time t_2, at which point the acknowledgment is sent back and delivered after t_1 time. So, after $2t_1 + t_2$ time responses from more than half processors are received, and in at most t_2 time the next step of the processor is scheduled when it again computes and communicates. This implies the following.

Claim 4.1.2. *For any algorithm, if the maximum expected number of* communicate *calls by any processor that the adversary can achieve is $O(T)$, then time complexity is also $O(T)$.*

Problem Statement: In *leader election* (*test-and-set*), each processor may return either *WIN* or *LOSE*. Every (correct) processor should return (*termination*), and only one processor may return *WIN* (*unique winner*). No processor may lose before the eventual winner starts its execution. The goal is to ensure that operations are *linearizable*, i.e., can be ordered such that (1) the first operation is *WIN* and every other return value is *LOSE*, and (2) the order of non-overlapping operations is respected.

[1]Note that the adversary can set t_1 or t_2 arbitrarily large, unknown to the algorithm, so the guarantees from the algorithm's prospective are still only that messages are *eventually* delivered and steps are *eventually* scheduled.

[2]Applied to asynchronous shared-memory, this yields an alternative definition of *step (time) complexity*, taking t_2 as an upper bound on the time for a thread to take a shared-memory step (and ignoring t_1). Counting all the delivery and non-trivial computation *steps* in message-passing gives an alternative definition of message complexity, corresponding to shared-memory *work complexity*.

4.2 Leader Election Algorithm

Our leader election algorithm guarantees that if k processors participate, the maximum expected number of communicate calls by any processor that the *strong adaptive* adversary can achieve is $O(\log^* k)$, and the maximum expected total number of messages is $O(nk)$. We start by illustrating the algorithmic idea from [Gel14].

4.2.1 The PoisonPill Technique

Consider the protocol specified in Figure 4-1 from the point of view of a participating processor. The procedure receives the id of the processor as an input, and returns a *SURVIVE*/*DIE* indication. All n processors react to received messages by replying with acknowledgments according to the communicate procedure. In the following, we

```
   Input: Unique identifier $i$ of the participating processor
   Output: $SURVIVE$ or $DIE$
   Local variables:
       $Status[n] = \{\perp\}$;
       $Views[n][n]$;
       int $coin$;
87 procedure PoisonPill$\langle i \rangle$
88     $Status[i] \leftarrow Commit$                              /* commit to coin flip */
89     communicate($propagate, Status[i]$)                          /* propagate status */
90     $coin \leftarrow$ random($1$ with probability $1/\sqrt{n}, 0$ otherwise)       /* flip coin */
91     if $coin = 0$ then $Status[i] \leftarrow Low\text{-}Pri$
92     else $Status[i] \leftarrow High\text{-}Pri$
93     communicate($propagate, Status[i]$)                  /* propagate updated status */
94     $Views \leftarrow$ communicate($collect, Status$)        /* collect status from $> n/2$ */
95     if $Status[i] = Low\text{-}Pri$ then
96         if $\exists proc.\, j : (\exists k : Views[k][j] \in \{Commit, High\text{-}Pri\}$ and
           $\forall k' : Views[k'][j] \neq Low\text{-}Pri)$ then
97             return $DIE$  /* $i$ has low priority, sees processor $j$ with either
               high priority or committed and not low priority, and dies */
98     return $SURVIVE$
```

Figure 4-1: PoisonPill Technique

call a *quorum* any set of more than $n/2$ processors.

Each participating processor announces that it is about to flip a random coin by moving to state *Commit* (lines 88-89), then obtain either low or high priority based on the outcome of a biased coin flip. The processor then propagates its priority information to a quorum (line 93). Next, it collects the status of other processors from a quorum using the communicate(*collect*, *Status*) call on line 94 that requests views of the array *Status* from each processor j, returning the set of replies received, of size at least $n/2$.

The crux of the round procedure is the *DIE* condition on line 97. A processor p returns *DIE* at this line if *both* of the following occur: (1) the processor p has low priority, *and* (2) it observes another processor q that does not have low priority in any of the views, but q has either high priority or is committed to flipping a coin (state *Commit*) in some view. Otherwise, processor p survives.

The key observations are that

Claim 4.2.1. *If all processors participating in* PoisonPill *return, at least one processor survives.*

We can also bound the maximum expected number of processors that survive:

Claim 4.2.2. *The maximum expected number of processors that return SURVIVE is* $O(\sqrt{n})$.

The proofs of these claims are given in [Gel14]. In the same work, applying this technique recursively with some extra care, an algorithm with an expected $O(\log\log n)$ time complexity is constructed. But we do not want to stop here.

4.2.2 Heterogeneous PoisonPill

Building a more efficient algorithm based on the PoisonPill technique requires reducing the number of survivors beyond $\Omega(\sqrt{n})$ without violating the invariant that not all participants may die. We control the coin flip bias, but setting the probability of flipping 1 to $1/\sqrt{n}$ is provably optimal. Let the adversary schedule processors to execute PoisonPill sequentially. With a larger probability of flipping 1, more than $\sqrt{n}$ processors are expected to get a high priority and survive. With a smaller probability, at least the first $\sqrt{n}$ processors are expected to all have low priority and survive. There are always $\Omega(\sqrt{n})$ survivors.

To overcome the above lower bound, after committing, we make each processor record the list ℓ of all processors including itself, that have a non-$\perp$ status in some view collected from the quorum. Then we use the size of list ℓ of a processor to determine its probability bias. Each processor also augments priority with its ℓ and propagates that as a status. This way, every time a high or low priority of a processor p is observed, ℓ of processor p is also known. Finally, the survival criterion is modified: each processor first computes set L as the union of all processors whose non-$\perp$ statuses it ever observed itself, and of the ℓ lists it has observed in priority informations in these statuses. If there is a processor in L for which no reported view has low priority, the current processor drops.

The algorithm is described in Figure 4-2. The particular choice of coin flip bias is influenced by factors that should become clear from the analysis. Despite modifica-

```
procedure HeterogeneousPoisonPill⟨$i$⟩
    $Status[i] \leftarrow \{.\mathsf{stat} = Commit, .\mathsf{list} = \{\}\}$          /* commit to coin flip */
    communicate($propagate, Status[i]$)                                  /* propagate status */
    $Views \leftarrow$ communicate($collect, Status$)                 /* collect status from $> n/2$ */
    $\ell \leftarrow \{j \mid \exists k : Views[k][j] \neq \perp\}$                    /* record participants */
    if $|\ell| = 1$ then $prob \leftarrow 1$                                          /* set bias */
    else $prob \leftarrow \frac{\log|\ell|}{|\ell|}$                                             /* set bias */
    $coin \leftarrow$ random($1$ with probability $prob, 0$ otherwise)               /* flip coin */
    if $coin = 0$ then $Status[i] \leftarrow \{.\mathsf{stat} = Low\text{-}Pri, .\mathsf{list} = \ell\}$   /* record priority
    and list */
    else $Status[i] \leftarrow \{.\mathsf{stat} = High\text{-}Pri, .\mathsf{list} = \ell\}$ /* record priority and list */
    communicate($propagate, Status[i]$)               /* propagate priority and list */
    $Views \leftarrow$ communicate($collect, Status$)                 /* collect status from $> n/2$ */
    if $Status[i].\mathsf{stat} = Low\text{-}Pri$ then
        $L \leftarrow \cup_{k,j: Views[k][j] \neq \perp} Views[k][j].\mathsf{list}$          /* union all observed lists */
        $L \leftarrow L \cup \{j \mid \exists k : Views[k][j] \neq \perp\}$            /* record new participants */
        if $\exists proc.\, j \in L : \forall k : Views[k][j].\mathsf{stat} \neq Low\text{-}Pri$ then
            return $DIE$       /* $i$ has low priority, learns about processor $j$
            participating whose low priority is not reported, and dies */
    return $SURVIVE$
```

Figure 4-2: Heterogeneous PoisonPill

tions, the same argument as in Claim 4.2.1 still guarantees at least one survivor. Let us now prove that the views of the processors have the following interesting *closure property*, which will be critical to bounding the number of survivors with low priority.

Claim 4.2.3. *Consider any set S of processors that each flip* 0 *and survive. Let U be the union of all L lists of processors in S. Then, for $p \in U$ and every processor q in the ℓ list of p, q is also in U.*

Proof. In order for processors in S to survive, they should have observed a low priority for each of the processors in their L lists. Thus, every processor $p \in U$ must flip 0, as otherwise it would not have a low priority. However, the low priority of p observed by a survivor was augmented by the ℓ list of p. According to the algorithm, the survivor includes in its own L all processors q from this ℓ list of p, implying $q \in U$. □

Next, let us prove a few other useful claims:

Claim 4.2.4. *If processor q completed executing line 101 no later than processor p completed executing line 101, then q will be included in the ℓ list of p.*

Proof. When p collects statuses on line 102 from a quorum, q is already done propagating its *Commit* on line 101. As every two quorum has an intersection, p will observe a non-$\perp$ status of q on line 103. □

Claim 4.2.5. *The probability of at least z processors flipping* 0 *and surviving is* $O(1/z)$.

Proof. Let S be the set of the z processors that flip 0 and survive and let us define U as in Claim 4.2.3. For any processor $p \in U$ and any processor q that completes executing line 101 no later than p, by Claim 4.2.4 processor q has to be contained in the ℓ list of p, which by the closure property (Claim 4.2.3) implies $q \in U$. Thus, if we consider the ordering of processors according to the time they complete executing line 101, all processors not in U must be ordered strictly after all processors in U.

Therefore, during the execution, first $|U|$ processors that complete line 101 must all flip 0. The adversary may influence the composition of U, but by the closure property, each ℓ list of processors in U contains only processors in U, meaning $|\ell| \leq |U|$. So the probability for each processor to flip 0 is at most $(1 - \frac{\log |U|}{|U|})$ and for all processors in U to flip 0's is at most $(1 - \frac{\log |U|}{|U|})^{|U|} = O(1/|U|)$. This is $O(1/z)$ since all z survivors from S are included in their own lists and hence also in U. □

We have never relied on knowing n. If $k \leq n$ processors participate in the heterogeneous PoisonPill, we get

Lemma 4.2.6. *The maximum expected number of processors that flip* 0 *and survive is* $O(\log k) + O(1)$.

Proof. Let Z denote the number of processors that flip 0 and survive. Then, $\mathbb{E}[Z] = \sum_{z=0}^{k} \Pr[Z \geq z]$ and $\Pr[Z \geq z] = O(1/z)$ by Claim 4.2.5. □

Lemma 4.2.7. *The maximum expected number of processors that flip* 1 *is* $O(\log^2 k) + O(1)$.

Proof. Consider the ordering of processors according to the time they complete executing line 101, breaking ties arbitrarily. Due to Claim 4.2.4, the processor that is ordered first always has $|l| \geq 1$, the second processor always computes $|l| \geq 2$, and so on. The probability of flipping 1 decreases as $|l|$ increases, and the best expectation achievable by adversary is $1 + \sum_{l=2}^{k} \frac{\log l}{l} = O(\log^2 k) + O(1)$ as desired. □

4.2.3 Final construction

The idea of implementing leader election is to have rounds of heterogeneous PoisonPill, where all processors participate in the first round and only the survivors of round r participate in round $r + 1$. Each processor p, before participating in round r_p, first propagates r_p as its current round number to a quorum, then collects information about the rounds of other processors from a quorum. Let R be the maximum round number of a processor in all views that p collected. To determine the winner, we use the idea from [SSW91]: if $R > r_p$, then p loses and if $R < r_p - 1$ then p wins. We also use a standard doorway mechanism [AGTV92] to ensure linearizability.

Figure 4-3 contains the pseudocode of PreRound procedure that processors execute before participating in round r. Every processor starts in the same initial non-negative round. The PreRound procedure takes round number r and the id of the processor as an input and outputs either *PROCEED*, *WIN* or *LOSE*. Each processor p first propagates r to a quorum, then collects information about the rounds of other processors also from a quorum. Let R be the maximum round number of a processor in all views that p collected. Using idea from [SSW91], if $R > r$, then p loses, if $R < r - 1$ then p wins and otherwise p returns *PROCEED*. To ensure linearizability we use a

```
    Input: Unique identifier $i$ of the participating processor, round number $r$
    Output: $PROCEED$, $WIN$, or $LOSE$
    Local variables:
        int $Round[n] = \{0\}$;
        int $Views[n][n]$;
        int $R$;
117 procedure PreRound$\langle i, r \rangle$
118     $Round[i] \leftarrow r$                                   /* record own round */
119     communicate($propagate, Round[i]$)                     /* propagate own round */
120     $Views \leftarrow$ communicate($collect, Round$)        /* collect round from $> n/2$ */
121     $R \leftarrow \max_{k,j|j \neq i}(Views[k][j])$         /* maximum round of other processors
        observed */
122     if $r < R$ then
123         return $LOSE$
124     if $R < r - 1$ then
125         return $WIN$
126     return $PROCEED$
```

Figure 4-3: PreRound procedure

standard doorway technique, described in Figure 4-4. This doorway mechanism is implemented by the variable *door* stored by the processors. A value false corresponds to the door being open and a value true corresponds to the door being closed. Each participating processor p starts by collecting the views of *door* from more than half of the processors on line 129. If a closed door is reported, p is too late and automatically returns *LOSE*. The door is closed by processors on line 132, and this information is then propagated to a quorum. The goal of the doorway is to ensure that no processor can lose *before* the winner has started its execution.

Finally we put the pieces together. Our complete leader election algorithm is described in Figure 4-5. It involves going through the doorway procedure in the

```
Output: $PROCEED$ or $LOSE$
Local variables:
    bool $door$ = false                                   /* door is initially open */
    bool $Doors[n]$;
procedure Doorway⟨⟩
    $Doors \leftarrow$ communicate($collect$, $door$)            /* collect $door$ from $> n/2$ */
    if $\exists j : Doors[j] =$ true then
        return $LOSE$                                 /* lose if the door is closed */
    $door \leftarrow$ true                                      /* close the door */
    communicate($propagate$, $door$)          /* propagates $door =$ true to $> n/2$ */
    return $PROCEED$
```

Figure 4-4: Doorway procedure

beginning, and then rounds of PreRound procedure possibly followed by participation in a HeterogeneousPoisonPill protocol for round r. Note that HeterogeneousPoisonPill protocols for different rounds are completely disjoint from each other.

```
Input: Unique identifier $i$ of the participating processor
Output: $WIN$ or $LOSE$
Local variables:
    int $r = 1$;
    $outcome$;
procedure LeaderElect⟨$i$⟩
    if Doorway⟨⟩ $= LOSE$ then
        return $LOSE$                                 /* lose if door was closed */
    repeat
        $outcome \leftarrow$ PreRound⟨$i, r$⟩                          /* preround routine */
        if $outcome \in \{WIN, LOSE\}$ then
            return $outcome$                          /* return if rounds permit */
        if HeterogeneousPoisonPill$_r$⟨i⟩ $= DIE$ then
            return $LOSE$               /* lose if did not survive the round */
        $r \leftarrow r + 1$
    until $never$
```

Figure 4-5: Leader election algorithm

We now prove the properties of the algorithm.

Lemma 4.2.8. *If all processors that call* LeaderElect *return, at least one processor returns WIN.*

Proof. Assume for contradiction that all processors that participate in PoisonPill return *LOSE*. Let us first prove that at least one processor always reaches the loop on line 138, or alternatively that not all processors can lose on line 137. This would mean that all processors return *LOSE* on line 131 of the Doorway procedure, but in that case the door would never be closed on line 132. Thus, all processor views would be $door = \mathsf{false}$, and no processor would actually be able to return on line 131.

Since we showed that at least one processor reaches the loop, let us consider the largest round r in which some processors return, either in the pre-round routine of round r on line 141 or because of the poison pill on line 143. By our assumption all these processors return *LOSE* in round r. But then, none of them may return on line 141, because this is only possible after returning *LOSE* on line 123, which only happens if a larger round than r is reported, contradicting our assumption that r is the largest round.

So, at least one processor participates in the $\mathsf{HeterogeneousPoisonPill}_r$ protocol. However, by exactly the same argument as in Claim 4.2.1, $\mathsf{HeterogeneousPoisonPill}_r$ is guaranteed to have at least one survivor which would then participate in round $r+1$, again contradicting that r is the largest round. □

Lemma 4.2.9. *At most one processor that executes* LeaderElect *can return WIN.*

Proof. A processor p can only return *WIN* from LeaderElect on line 141, which only happens after p returns *WIN* from PreRound call with some round r. This means p first propagated round r to a quorum on line 119, then collected views of *Round* array on line 120, and observed maximum round $R < r-1$ of any processor in any of the views. This implies that when p finished propagating r to a quorum, no processor had finished propagating $r-1$, i.e. executing line 119 in round $r-1$. Otherwise, since every two quorums have an intersection, p would have observed round $r-1$ and $R < r-1$ would not hold. But for every other processor q, when q executes line 120 in round $r-1$ and invokes the PreRound procedure, R will be at least r since p has already propagated to a quorum, so q will observe $r-1 < r$ and return *LOSE* on line 123 and subsequently return *LOSE* from LeaderElect. □

Lemma 4.2.10. *Our leader election algorithm is linearizable.*

Proof. All processors that execute LeaderElect cannot return $LOSE$ by Lemma 4.2.8. Therefore, in every execution we can find LeaderElect invocation where processor either does not return, or returns WIN. On the other hand, by Lemma 4.2.9, no more than one processor can return WIN. If no processor returns WIN, let us linearize the processor that invoked LeaderElect the earliest as the leader. This way, we always have an unique processor to be linearized as the winner. We linearize it at the beginning of its invocation interval, say point P, and claim that every remaining LeaderElect call can be linearized as returning $LOSE$ after P.

Assume contrary, then the problematic LeaderElect invocation must return before P, and we know it has to return $LOSE$. By definition, this earlier call either closes the door or observes a closed door while executing the Doorway procedure. Therefore, the later call that we are linearizing as the winner has to observe a closed door on line 129 and cannot avoid returning $LOSE$ on line 131. Hence, this invocation can never return WIN, and since we are linearizing it as winner, it should be the case that it does not return and no other processor returns WIN. We picked this invocation to have the earliest starting point, so every other LeaderElect invocation that does not return must start after P. Let us now consider an extension of the current execution where the processors executing these invocations are continuosly scheduled to take steps and all messages are delivered. According to the above argument, since all invocations start after P, these processors must observe a closed door on line 129 and return $LOSE$ after only finitely many steps. We have hence constructed a valid execution where all processors that execute LeaderElect return $LOSE$. This contradiction with Lemma 4.2.8 completes the proof. □

We need one final claim before proving the main theorem.

Claim 4.2.11. *The maximum expected number of participants decreases at least by some fixed constant fraction in every two rounds.*

Proof. This obviously holds for a single participant, because it will return WIN in the next round and the number of participants after that will be zero.

We know that for k participants in some round, by Lemma 4.2.6 and Lemma 4.2.7, the maximum expected number of participants in the next round is $O(\log^2 k + 1)$. This implies that for a large enough constant D, there is constant $c_1 < 1$ such that for $k > D$ the maximum expected number of participants in the next round, and thus in all rounds thereafter, is at most $c_1 k$. If $k \leq D$, then the first processor that finishes executing line 101 flips 1 with at least a constant probability. In this case, all processors that flip 0 will die, and the expected number of the remaining processors that flip 0 is at least $\frac{k-1}{2} \leq \frac{k}{4}$ for $k \geq 2$. This is because the expected number of remaining processors that flip 1 is at most $\frac{k-1}{2}$, as each of them observes at least the first processor and itself, hence has no more than 1/2 probability of flipping 1. Thus, if $k \leq D$, with a constant probability, a constant fraction of participants dies, meaning that there is a constant $c_2 < 1$ such that the maximum expected number of participants is at most $c_2 k$. Setting $c = \max(c_1, c_2) < 1$ we obtain that the maximum expected number of participants in every two rounds always decreases by at least a constant fraction to ck. □

Theorem 4.2.12. *Our leader election algorithm is linearizable. If there are at most* $\lceil n/2 \rceil - 1$ *processor faults, all non-faulty processors terminate with probability* 1. *For* k *participants, it has time complexity* $O(\log^* k)$ *and message complexity* $O(kn)$.

Proof. We have shown linearizability in Lemma 4.2.10.

All $k \geq 1$ processors participate in the first round. The maximum expected number of processors that participate in round 3 is clearly no more than the maximum expected number of survivors of the first round, which by Lemma 4.2.6 and Lemma 4.2.7 for $k > 1$ can be written as $C(\log^2 k + 2\log k)$ for some constant C. If $k = 1$, then this lone processor will observe all other processors in round 0, leading to $R = 0$ and as current round is $r = 2$ it will return *WIN* in the second round. Hence, for $k = 1$, there will be zero participants in the third round. Thus, for any k, the maximum expected number of participants in round 3 is at most $f(k) = C(\log^2 k + 2\log k)$.

Let us say the adversary can achieve a probability distribution for round 3 such that there are K_i participants with probability p_i. We have shown above that

$$\sum_i p_i K_i \leq f(k) \tag{4.2.1}$$

Now, using the same argument as above, we can bound the maximum expected number of participants in round 5 to be at most $\sum p_i f(K_i)$. Function f is concave for non-negative arguments, and for arguments larger than a constant it is also monotonically increasing. This implies that either $\sum p_i K_i$, the expected number of participants in round 3, is constant, or

$$\sum p_i f(K_i) \leq f(\sum p_i K_i) \leq f(f(k)) \tag{4.2.2}$$

where the first part is Jensen's inequality and the second follows from (4.2.1) and the monotonicity property. Similarly, we get that unless the maximum expected number of participants in round 5 is less than a constant, the maximum number of participants in round 7 is at most $f(f(f(k)))$, and so on. Since $f(f(k)) \geq \log k$ for all k larger than some constant, if we denote by S_0 the number of participants in round $1 + 2\log^* k$, maximum $\mathbb{E}[S_0]$ that the adversary can achieve must also be constant. These S_0 participants execute the same algorithm, with S_1 of them participating in the next round, etc.

Let R be the number of remaining rounds. Expectation of R can be written as

$$\mathbb{E}[R] = \sum_{i=1}^{\infty} \Pr[R \geq i] = \sum_{i=1}^{\infty} \Pr[S_i \geq 1] \leq \sum_{i=1}^{\infty} \mathbb{E}[S_i] \tag{4.2.3}$$

where the equality is by the definition of rounds and then we apply Markov's inequality to get to expectations. Finally, by Claim 4.2.11 we get that $\mathbb{E}[R] = O(\mathbb{E}[S_0]) = O(1)$ and thus the maximum total number of rounds any processor participates in is $O(\log^* k)$, and processors perform only fixed, constantly many communicate calls per round. Time complexity follows from Claim 4.1.2.

To bound the maximum expected number of messages, let Q_r be the number of participants in round r, counting from the very first round. Since each processor sends $O(n)$ messages per round, the maximum expected number of messages is $\sum_{r=1}^{\infty} \mathbb{E}[O(nQ_r)] = n \cdot \mathbb{E}[O(Q_1)] = O(nk)$ using Claim 4.2.11.

If there are at most $\lceil n/2 \rceil - 1$ processor faults, all communicate calls return, and processors must enter larger rounds. However, the probability that all processors terminate before reaching round r is $1 - \Pr[Q_r \geq 1] \geq 1 - \mathbb{E}[Q_r]$ which tends to 1 as r increases by Claim 4.2.11. $\square$

Chapter 5

Conclusions

We make progress on understanding the complexity of fundamental tasks in standard distributed models. Our work leads to numerous new interesting open problems and directions for future research.

Population Protocols: Our lower bounds can be seen as bad news for algorithm designers, since it show that stabilization is slow even if the protocol implements a super-constant number of states per node. On the positive side, the achievable stabilization time improves quickly as the size of the state space nears the lower bound thresholds. However, this still motivates the unresolved problem of establishing the optimal state complexity of algorithms that converge fast (in polylogarithmic parallel time). Our lower bounds on majority and leader election apply to algorithms that stabilize fast, which is a stronger requirement than convergence. For majority our results highlight a separation. While fast stabilization requires $\Omega(\log n)$ states, it is possible to design an algorithm that would converge fast using $O(\log \log n)$ states. While some of the same technical tools may well turn out to be useful when dealing with the convergence requirement as opposed to stabilization, solving this problem is likely to require developing novel and interesting techniques.

Extending the results about leader election and majority to other tasks is another important direction. [BDS17] takes a step in this direction, exploring the complexity of computing different types of predicates.

Finally, a technical challenge that we would love to see settled is getting rid of the "output dominance" assumption in our majority lower bound. We conjecture that the same $\Omega(\log n)$ lower bound must hold unconditionally (showing otherwise would also be a an impressive and surprising result). Proving this is likely to involve more complex"surgeries" on transition sequences.

Complexity-Based Hierarchy: We defined a hierarchy based on the space complexity of solving consensus. We used consensus because it is a well-studied, general task that seems to capture a fundamental difficulty of multiprocessor synchronization. Moreover, consensus is *universal*: any sequentially defined object can be implemented in a wait-free way using only consensus objects and registers [Her91].

We did not address the issue of universality within our hierarchy. One history object can be used to implement any sequentially defined object. Consequently, it may make sense to consider defining a hierarchy on sets of instructions based on their ability to implement a history object, a compare-and-swap object, or a repeated consensus object shared by n processes. However, the number of locations required for solving n-consensus is the same for implementing these long-lived objects, for many of the instruction sets that we considered. (It appears that this may not be true for $\{read, increment\}$.)

It may be that a truly accurate complexity-based hierarchy would have to take time complexity into consideration. It is immediately unclear which definition of time to use. Exploring this may be an important future direction. One reasonable candidate is the *solo step complexity*, i.e. maximum number of solo steps that any process needs to ever take to return a value. However, getting tight bounds on this complexity measure is challenging (see [ABBH16] for recent progress on the solo step complexity of consensus), maybe more so than for space complexity.

It is standard to assume that memory locations have unbounded size, in order to focus solely on the challenges of synchronization. For a hierarchy to be truly practical, however, we might need to consider the size of the locations used by an algorithm.

There are several other interesting open problems. To the best of our knowledge, all existing space lower bounds rely on a combination of covering and indistinguishability arguments. However, when the covering processes apply *swap*(x), as opposed to *write*(x), they can observe differences between executions, so they can no longer be reused to maintain indistinguishability. This means that getting a tighter space lower bound for $\{\mathit{swap}(x), \mathit{read}()\}$ would most likely require a completely novel approach. An algorithm that uses less than $n-2$ space would be even more surprising, as the processes would necessarily have to adapt their access patterns to the memory locations based on the swapped values, in order to circumvent the argument from [Zhu16]. The authors are unaware of any such algorithm.

We conjecture that, for sets of instructions, $\mathcal{I}$, which contain only *read*$()$, *write*(x), and either *increment*$()$ or *fetch-and-increment*$()$, $\mathcal{SP}(\mathcal{I}, n) \in \Theta(\log n)$. Similarly, we conjecture, for $\mathcal{I} = \{\mathit{read}(), \mathit{write}(0), \mathit{write}(1)\}$, $\mathcal{SP}(\mathcal{I}, n) \in \Theta(n \log n)$. Proving this is likely to require techniques that depend on the number of input values, such as in the lower bound for m-valued adopt-commit objects in [AE14a].

We would like to understand the properties of sets of instructions at certain levels in the hierarchy. For instance, what properties enable a collection of instructions to solve n-consensus using a single location? Is there an interesting characterization of the sets of instructions $\mathcal{I}$ for which $\mathcal{SP}(\mathcal{I}, n)$ is constant?

How do subsets of a set of instructions relate to one another in terms of their locations in the hierarchy? Alternatively, what combinations of sets of instructions decrease the amount of space needed to solve consensus? For example, using only *read*$()$, *write*(x), and either *increment*$()$ or *decrement*$()$, more than one memory location is needed to solve binary consensus. But with both *increment*$()$ and *decrement*$()$ a single location suffices. Are there general properties governing these relations?

Our practical implementation also motivates finding a compact set of synchronization instructions that, when supported, is equally powerful as the set of instructions used today as an important question to establish in future research.

Simulation: As a corollary of our lower bound for k-obstruction-free k-set agreement, we get a right lower bound of n registers for consensus, which further emphasizes that our simulation-based technique has strong potential to prove previously elusive results. The simulation technique can be generalized to x-obstruction-free algorithms for any $x \leq k$ [EGZ17]. The lower bound for $x = 1$ of n/k registers also applies to nondeterministic solo terminating algorithms and thus, to randomized wait-free algorithms. We are also able to prove a space lower bound of $\lfloor n/2 \rfloor + 1$ for the ϵ-agreement task for sufficiently small ϵ. ϵ-agreement requires processes to return values at most ϵ apart from each other. Our goal is to get a general characterization of tasks for which this approach can be applied systematically to get tight or close to tight space lower bounds. Taking this even further, it is intriguing to explore whether similar simulation technique can be used for other complexity measures, for instance solo step complexity, or randomized step (time) complexity.

We believe that the simulation technique can be further improved. For instance, in our constructions, simulators do not share simulated processes. One could envision using BG simulation [BG93], which might allow getting stronger space lower bounds.

Finally, we introduce the informal concept of local and global proofs, and a formal *boundary condition*. Exploring the complexity of tasks under this condition might allow us to make progress and get insight on more problems than formerly possible.

Message Passing: Expected time complexity of leader election in asynchronous message passing, and in fact, also in asynchronous shared memory against the adaptive adversary is one of the most notorius open problems in the field. If we replace collects by atomic snapshots, our algorithm also works in shared memory and uses no more than expected $\log^\star k$ snapshot (and write) steps for k participants. Thus, it shows that if we assume unit-cost snapshots in shared-memory for the purposes of proving a lower bound (which is a natural technical step), then the best lower bound we can aim for is $\log^\star n$.

Bibliography

[AA11] Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Proceedings of 25th International Symposium on Distributed Computing*, DISC '11, pages 97–109, 2011.

[AACH+14] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *Journal of the ACM (JACM)*, 61(3):18:1–18:51, May 2014.

[AAD+93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[AAD+06] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed computing*, 18(4):235–253, March 2006.

[AAE08a] Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, September 2008.

[AAE08b] Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, July 2008.

[AAE+17] Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *Pro-*

ceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms, SODA '17, pages 2560–2579, 2017.

[AAER07] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, November 2007.

[AAG$^+$10] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of 24th International Symposium on Distributed Computing*, DISC '10, pages 94–108, 2010.

[AAG17] Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. *arXiv preprint arXiv:1704.04947*, 2017.

[ABBH16] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Lower bound on the step complexity of anonymous binary consensus. In *Proceedings of the 30th International Symposium on Distributed Computing*, DISC '16, pages 257–268, 2016.

[ABGG12] Dan Alistarh, Michael A. Bender, Seth Gilbert, and Rachid Guerraoui. How to allocate tasks asynchronously. In *Proceedings of the 53rd IEEE Symposium on Foundations of Computer Science*, FOCS '12, pages 331–340, 2012.

[ABKU99] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[ABND$^+$90] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *Journal of The ACM (JACM)*, 37(3):524–548, July 1990.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[Abr88] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 291–302, 1988.

[AC08] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM (JACM)*, 55(5):20:1–20:26, October 2008.

[AC11] Hagit Attiya and Armando Castañeda. A non-topological proof for the impossibility of k-set agreement. In *Proceedings of the 13th Symposium on Self-Stabilizing Systems*, SSS '11, pages 108–119, 2011.

[AE14a] James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3):451–474, 2014.

[AE14b] Hagit Attiya and Faith Ellen. Impossibility results for distributed computing. *Synthesis Lectures on Distributed Computing Theory*, 5(1):1–162, 2014.

[AG15] Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*, ICALP '15, pages 479–491, 2015.

[AGM02] Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.

[AGTV92] Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms*, WDAG '92, pages 85–94, 1992.

[AGV15a] Dan Alistarh, Rati Gelashvili, and Adrian Vladu. How to elect a leader faster than a tournament. In *Proceedings of the 34th ACM Symposium* *on Principles of Distributed Computing*, PODC '15, pages 365–374, 2015.

[AGV15b] Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. Fast and exact majority in population protocols. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 47–56, 2015.

[AH90] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

[AP12] Hagit Attiya and Ami Paz. Counting-based impossibility proofs for renaming and set agreement. In *Proceedings of the 26th International Symposium on Distributed Computing*, DISC '12, pages 356–370, 2012.

[AR16] Alexandr Andoni and Ilya Razenshteyn. Tight lower bounds for data-dependent locality-sensitive hashing. In *Proceedings of the 32nd International Symposium on Computational Geometry*, SoCG '16, pages 9:1–9:11, 2016.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[BB04] James M Bower and Hamid Bolouri. *Computational modeling of genetic and biochemical networks.* MIT press, 2004.

[BCER17] Andreas Bilke, Colin Cooper, Robert Elsaesser, and Tomasz Radzik. Population protocols for leader election and exact majority with $O(\log^2 n)$ states and $O(\log^2 n)$ convergence time. *arXiv preprint arXiv:1705.01146*, 2017.

[BCSV06] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing*, 35(6):1350–1385, 2006.

[BDS17] Amanda Belleville, David Doty, and David Soloveichik. Hardness of computing and approximating predicates and functions with leaderless population protocols. In *Proceedings of the 44nd International Colloquium on Automata, Languages, and Programming*, ICALP '17, 2017.

[BFK$^+$16] Petra Berenbrink, Tom Friedetzky, Peter Kling, Frederik Mallmann-Trenn, and Chris Wastell. Plurality consensus via shuffling: Lessons learned from load balancing. *arXiv preprint arXiv:1602.01342*, 2016.

[BG93] Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, STOC '93, pages 91–100, 1993.

[BG97] Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 189–198, 1997.

[BKRS96] Jonathan F. Buss, Paris C. Kanellakis, Prabhakar L. Ragde, and Alex A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20:45–86, January 1996.

[BL93] James E Burns and Nancy A Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

[BMW$^+$13] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 325–340, 2013.

[BO83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the*

2nd ACM Symposium on Principles of Distributed Computing, PODC '83, pages 27–30, 1983.

[Bow11] Jack Bowman. Obstruction-free snapshot, obstruction-free consensus, and fetch-and-add modulo k. Technical Report TR2011-681, Dartmouth College, Computer Science, Hanover, NH, 2011.

[BRS15] Zohir Bouzid, Michel Raynal, and Pierre Sutra. Brief announcement: Anonymous obstruction-free (n, k)-set agreement with n- k+ 1 atomic read/write registers. In *Proceedings of the 29th International Symposium on Distributed Computing*, DISC '15, page 669, 2015.

[CCDS15] Ho-Lin Chen, Rachel Cummings, David Doty, and David Soloveichik. Speed faults in computation by chemical reaction networks. *Distributed Computing*, 2015. To appear.

[CCN12] Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Nature Scientific Reports*, 2:656:1–656:9, September 2012.

[CDS$^+$13] Yuan-Jyue Chen, Neil Dalchau, Niranjan Srnivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from dna. *Nature Nanotechnology*, 8(10):755–762, September 2013.

[CDS14] Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural computing*, 13(4):517–534, 2014.

[Cha93] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

[CKL16] Luca Cardelli, Marta Kwiatkowska, and Luca Laurenti. Programming discrete distributions with chemical reaction networks. In *Proceedings* *of the 22nd International Conference on DNA Computing and Molecular Programming*, DNA22, pages 35–51. Springer, 2016.

[CMN$^+$11] Ioannis Chatzigiannakis, Othon Michail, Stavros Nikolaou, Andreas Pavlogiannis, and Paul G Spirakis. Passively mobile communicating machines that use restricted space. In *Proceedings of the 7th ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing*, FOMC '11, pages 6–15, 2011.

[Dav04] Matei David. Wait-free linearizable queue implementations, 2004.

[DGFGR13] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Black art: Obstruction-free k-set agreement with |mwmr registers| < |processes|. In *Proceedings of the 1st International Conference on Networked Systems*, NETYS '13, pages 28–41, 2013.

[DGFKR15] Carole Delporte-Gallet, Hugues Fauconnier, Petr Kuznetsov, and Eric Ruppert. On the space complexity of set agreement. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 271–280, 2015.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[DMST07] Ian B. Dodd, A. M. Micheelsen, Kim Sneppen, and Geneviéve Thon. Theoretical analysis of epigenetic cell memory by nucleosome modification. *Cell*, 129(4):813–822, 2007.

[Dot14] David Doty. Timing in chemical reaction networks. In *Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 772–784, 2014.

[DS97] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997.

[DS15] David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. In *Proceedings of the 29th International Symposium on Distributed Computing*, DISC '15, pages 602–616, 2015.

[DV12] Moez Draief and Milan Vojnovic. Convergence speed of binary interval consensus. *SIAM Journal on Control and Optimization*, 50(3):1087–1109, May 2012.

[EFR08] Faith Ellen, Panagiota Fatourou, and Eric Ruppert. The space complexity of unbounded timestamps. *Distributed Computing*, 21(2):103–115, 2008.

[EGSZ16] Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. A complexity-based hierarchy for multiprocessor synchronization:[extended abstract]. In *Proceedings of the 35th ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 289–298, 2016.

[EGZ17] Faith Ellen, Rati Gelashvili, and Leqi Zhu. Enter the simulation: Space lower bounds for agreement. `http://www.cs.toronto.edu/~lezhu/enter-the-simulation.pdf`, 2017. Manuscript.

[FHS98] Faith Ellen Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM (JACM)*, 45(5):843–862, 1998.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[FLMS05] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC '05, pages 78–92, 2005.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985.

[Gel14] Rati Gelashvili. Leader election and renaming with optimal message complexity. Master's thesis, Massachusetts Institute of Technology, 2014.

[Gel15] Rati Gelashvili. On the optimal space complexity of consensus for anonymous processes. In *Proceedings of the 29th International Symposium on Distributed Computing*, DISC '15, pages 452–466, 2015.

[GHHW13] George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An $\mathcal{O}(\sqrt{n})$ space bound for obstruction-free leader election. In *Proceedings of the 27th International Symposium on Distributed Computing*, DISC '13, pages 46–60, 2013.

[GHHW15] George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. Test-and-set in optimal space. In *Proceedings of the 47th ACM Symposium on Theory of Computing*, STOC '15, pages 615–623, 2015.

[GHKR16] Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. Read-write memory and k-set consensus as an affine task. In *Proceedings of the 20th International Conference on Principles of Distributed Systems*, OPODIS '16, pages 6:1–6:17, 2016.

[GKM14] Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In *Proceedings of the 33th ACM symposium on Principles of Distributed Computing*, PODC '14, pages 222–231, 2014.

[GKSW17] Rati Gelashvili, Idit Keidar, Alexander Spiegelman, and Roger Wattenhofer. Towards reduced instruction sets for synchronization. *arXiv preprint arXiv:1705.02808*, 2017.

[GP16] Mohsen Ghaffari and Merav Parter. A polylogarithmic gossip algorithm for plurality consensus. In *Proceedings of the 35th ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 117–126, 2016.

[GR05] Rachid Guerraoui and Eric Ruppert. What can be implemented anonymously? In *Proceedings of the 19th International Symposium on Distributed Computing*, DISC '05, pages 244–259. 2005.

[GS17] Leszek Gąsieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. *arXiv preprint arXiv:1704.07649*, 2017.

[GW12a] George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 19–28, 2012.

[GW12b] George Giakkoupis and Philipp Woelfel. A tight rmr lower bound for randomized mutual exclusion. In *Proceedings of the 44th ACM Symposium on Theory of Computing*, STOC '12, pages 983–1002, 2012.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.

[HHPW14] Maryam Helmi, Lisa Higham, Eduardo Pacheco, and Philipp Woelfel. The space complexity of long-lived and one-shot timestamp implementations. *Journal of the ACM (JACM)*, 61(1):7, 2014.

[HKR13] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed computing through combinatorial topology.* Newnes, 2013.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings* *of the 23rd IEEE International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, 2003.

[HR00] Maurice Herlihy and Eric Ruppert. On the existence of booster types. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*, FOCS '00, pages 653–663, 2000.

[HS99] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of The ACM (JACM)*, 46(6):858–923, November 1999.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[HW09] Danny Hendler and Philipp Woelfel. Randomized mutual exclusion in o(log n/log log n) rmrs. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 26–35, 2009.

[HW10] Danny Hendler and Philipp Woelfel. Adaptive randomized mutual exclusion in sub-logarithmic expected time. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, PODC '10, pages 141–150, 2010.

[Int12] Intel. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, 2012. Manual.

[Jay93] Prasad Jayanti. On the robustness of herlihy's hierarchy. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 145–157, 1993.

[KLMadH92] Richard M Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, STOC '92, pages 318–326, 1992.

[KMPS95] Anil Kamath, Rajeev Motwani, Krishna Palem, and Paul Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. *Random Structures & Algorithms*, 7(1):59–80, August 1995.

[KS92] Paris C. Kanellakis and Alex A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5(4):201–217, April 1992.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LH00] Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM Journal on Computing*, 30(3):689–728, 2000.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MA13] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 48 of *PPoPP '13*, pages 103–112, 2013.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[McD89] Colin McDiarmid. On the method of bounded differences. *Surveys in Combinatorics*, 141(1):148–188, 1989.

[MNRS14] George B. Mertzios, Sotiris E. Nikoletseas, Christoforos Raptopoulos, and Paul G. Spirakis. Determining majority in networks with local interactions and very small local memory. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming*, ICALP '14, pages 871–882, 2014.

[PAC$^+$97] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, 1997.

[PR01] Rasmus Pagh and Flemming Friche Rodler. *Cuckoo hashing*. Springer, 2001.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, April 1980.

[PTW15] Yuval Peres, Kunal Talwar, and Udi Wieder. Graphical balanced allocations and the $(1 + \beta)$-choice process. *Random Structures & Algorithms*, 47(4):760–775, July 2015.

[PVV09] Etienne Perron, Dinkar Vasudevan, and Milan Vojnovic. Using three states for binary consensus on complete graphs. In *Proceedings of the 28th IEEE Conference on Computer Communications*, INFOCOM '09, pages 2527–2535, 2009.

[Ray12] Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[Rup00] Eric Ruppert. Determining consensus numbers. *SIAM Journal on Computing*, 30(4):1156–1168, 2000.

[Sch97] Eric Schenk. The consensus hierarchy is not robust. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, PODC '97, page 279, 1997.

[SHG16] Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In *Proceedings of the 30th International Symposium on Distributed Computing*, DISC '16, pages 428–441, 2016.

[Sol49] Lefschetz Solomon. *Introduction to Topology.* Princeton University Press, 1949.

[SP89] Eugene Styer and Gary L Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, PODC '89, pages 177–191, 1989.

[SSW91] Michael Saks, Nir Shavit, and Heather Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, SODA '91, pages 351–362, 1991.

[SZ00] Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, 2000.

[Tau06] Gadi Taubenfeld. *Synchronization algorithms and concurrent programming.* Pearson Education, 2006.

[Tau17] Gadi Taubenfeld. Contention-sensitive data structures and algorithms. *Theoretical Computer Science*, 2017.

[Tho79] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, June 1979.

[TWS15] Chris Thachuk, Erik Winfree, and David Soloveichik. Leakless dna strand displacement systems. In *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming*, DNA21, pages 133–153. Springer, 2015.

[YNG98] Jiong Yang, Gil Neiger, and Eli Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 297–306, 1998.

[Zhu15] Leqi Zhu. Brief announcement: Tight space bounds for memoryless anonymous consensus. In *Proceedings of the 29th International Symposium on Distributed Computing*, DISC '15, page 665, 2015.

[Zhu16] Leqi Zhu. A tight space bound for consensus. In *Proceedings of the 48th ACM Symposium on Theory of Computing*, STOC '16, pages 345–350, 2016.