Fundamental Bounds and Algorithms in Distributed Reliable Storage

Research Thesis

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

Alexander Spiegelman

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

Sivan 5778 Haifa June 2018
The research thesis was done under the supervision of Prof. Idit Keidar in the Viterbi Faculty of Electrical Engineering, Technion – Israel Institute of Technology, Haifa, Israel.

The generous financial help of the Technion and the Azrieli Fellowship is gratefully acknowledged.

I wish to thank the Wolf Foundation, Dr. Jacob Isler Foundation, and the Meyer academic excellence program for their awards.
This research thesis is dedicated to my mother who gave up on her Ph.D. in order to give me birth.
Acknowledgment

I would like to sincerely thank the many people who made this research fruitful and enjoyable.

- I wish to thank my advisor Idit Keidar for guiding me and teaching me how to do research. I thank you for your endless support and for being such an inspiring role model. Your advice is priceless, and your caring is admirable. It was a great pleasure to work with you and to learn from you.

- I wish to thank Ittai Abraham, Eddie Bortnikov, Prof. Yuval Cassuto, Prof. Gregory Chockler, Rati Gelashvili, Guy Golan Gueta, Eshcar Hillel, Heidi Howard, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Prof Moshe Tennenholtz for very productive and enjoyable collaborations during my Ph.D. In particular, I wish to thank Prof. Yuval Cassuto, Prof. Gregory Chockler and Dahlia Malkhi, who are my co-authors on papers my thesis is based on, for your help, support, and guidance.

- I would like to thank our research group’s team members for the fun time we spent together, and the ideas and feedbacks we shared. I enjoyed our group meetings, the lunch times talks, and the professional discussions. In particular, I would like to thank Kfir Lev-Ari and Alon Berger for the discussions we had and the work we did together during my Ph.D. And of course, I wish to thank Naama Kraus for being such a wonderful office-mate.

- I wish to thank Orly Babad-Tamir and Danit Cohen for their administrative support. Thank you for your dedication and your efficient help in any need - you helped to make this journey pleasant.

- I wish to thank my parents, Olga Klin and Ilya Spiegelman, for raising me with wisdom, always believing in me, and pushing me forward without pressuring.

- I thank my brother Roman Spiegelman for his interest, support, and encouragement.

- I wish to thank my parents in law, Dimitry Nouzman and Irina Kaloujski for their kind support. I am grateful for the assistance, which enabled me to invest time and resources in my research.

- To my beloved baby, Shaked, who gave and gives me happiness. I thank you for never crying at nights before deadlines.)

- And most importantly, I wish to thank my beloved wife Yelizabetha (Liza) Nouzman. Your support and sacrifice made this research possible. Thank you for being there for me every time I needed, I am grateful and fortunate to have you by my side.
List of Publications

Publications this thesis is based on


Other publications


Technical reports

# Table of Contents

Abstract 1

List of Acronyms 3

List of Symbols 4

1 Introduction 5
   1.1 Storage cost 5
   1.2 Dynamic reconfiguration 7

2 The Power of Primitives for Fault-Tolerant Register Emulations 9
   2.1 Model 13
      2.1.1 Shared Objects 13
      2.1.2 Registers 13
      2.1.3 Consistency Conditions 14
      2.1.4 System Model 14
      2.1.5 Properties of the Emulation Algorithms 15
   2.2 A Max-Register Emulation With One CAS 15
      2.2.1 Correctness 16
   2.3 Resource Complexity of Write-Sequential k-register Emulation 18
      2.3.1 Lower bound overview 18
      2.3.2 Lower Bounds 19
      2.3.3 Upper Bound 28
   2.4 Discussion and Future Directions 33

3 Space Bounds for Reliable Storage: Fundamental Limits of Coding 34
   3.1 Model 36
      3.1.1 Preliminaries 36
      3.1.2 Storage algorithm model and assumptions 38
   3.2 Related work 40
   3.3 Storage Lower Bound 41
   3.4 Adaptive Regular Register 47
TABLE OF CONTENTS

3.4.1 Algorithm ......................................................... 47
3.4.2 Correctness Proofs .............................................. 51
3.5 A (Simple) Safe and Wait-free Algorithm ......................... 55
  3.5.1 Algorithm ..................................................... 56
  3.5.2 Correctness proof ............................................. 56
3.6 Discussion ......................................................... 58

4 On Liveness of Dynamic Storage ...................................... 59
  4.1 Model ............................................................ 62
    4.1.1 Preliminaries ............................................... 62
    4.1.2 Dynamic storage ........................................... 62
  4.2 Impossibility of Wait-Free Dynamic Safe Storage ............... 65
  4.3 Oracle-Based Dynamic Atomic Storage ......................... 66
    4.3.1 Dynamic failure detector .................................. 67
    4.3.2 Dynamic storage algorithm ............................... 68
    4.3.3 Correctness proof ......................................... 74
  4.4 Conclusion ..................................................... 79

5 Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution ............................................ 80
  5.1 Related Work ................................................... 83
  5.2 Dynamic Model .................................................. 84
  5.3 Reconfiguration Abstraction .................................... 86
  5.4 Building Dynamic Objects Using Reconfiguration ............... 88
    5.4.1 Dynamic atomic read/write register ..................... 88
    5.4.2 Dynamic atomic max-register ........................... 90
    5.4.3 Read/write register correctness proof .................. 91
  5.5 The Reconfiguration Abstraction Implementation ............... 94
    5.5.1 CoS building block ..................................... 94
    5.5.2 Simple Reconfiguration .................................. 96
    5.5.3 Optimal Reconfiguration ................................. 98
    5.5.4 Reconfiguration Correctness Proof ...................... 100
    5.5.5 Reconfiguration Complexity Proof ...................... 103
  5.6 Conclusions ..................................................... 110

6 Conclusion .................................................................. 112

Bibliography .................................................................. 114
  1 Additional results of Chapter 2 ................................. 120
List of Figures

1.1 Encoding and decoding using xor. ........................................ 6

2.1 An illustration of the runs constructed in Lemma 5. ................. 25

2.2 A possible mapping from R to S in case n = 6, k = 5, and f = 2. .... 29

3.1 Clients and base objects. .................................................. 36

3.2 A model for code-based storage. Encoding and decoding are captured by oracles. .............................................................. 39

3.3 Black-box coding. Runs r and r_v have the same trace except that write w is invoked with u in r and with v in r_v, and each base object b_0's state (blocks and meta-data) is identical at all times in both runs, except that blocks produced by w's oracle in r are replaced in r_v by the corresponding blocks of v. ......................................................... 40

3.4 Example scenario in run of a storage algorithm with adversary Ad. In this example, 2D/5 < \ell < D. At time t, only w_2 and w_4 are in C^-(t), where w_4 has no pending RMWs and w_2 has one triggered RMW on b_1 \in F(t) and one triggered RMW on b_3 \notin F(t). Therefore, by the first rule, Ad schedules the response on the RMW triggered by w_2 on b_3. In this example w_2 overwrites w_3's block in b_3, thus w_3 moves from C^- to C^-.

Then, at time t + 1, no response can be scheduled by rule 1 (no operation in C^- (t + 1) has a pending RMW on a base object in N \ F(t + 1)), so by rule 2, Ad chooses w_2 and lets it trigger an RMW on base object b_2. Now since w_2 is the only operation that has a pending RMW on a base object not in F(t + 2), Ad schedules the response on the RMW triggered by w_2 on b_2 at time t + 2. In this example w_2 adds a block with \ell bits to b_2. Thus, c_2 is included in C^+(t + 3). In addition, b_2 stores more than \ell bits at time t + 3, so it belongs to F(t + 3). ......................................................... 44

4.1 Notation illustration. add(p) (remove(p)) represents reconfig(\langle add, p \rangle) (respectively, reconfig(\langle remove, p \rangle)). ......................................................... 63

4.2 Illustration of the infinite run r ............................................ 67
4.3 Flow illustration: process $p_2$ is slow. After stabilization time, process $p_1$ helps it by proposing its operation. Once $p_2$’s operation is decided, it is reflected in every up-to-date $sm$. Therefore, even if $p_1$ fails before informing $p_2$, $p_2$ receives from the next process that performs an operation, namely $p_3$, an $sm$ that reflects its operation, and thus returns. Line arrows represent messages, and block arrows represent operation or consensus invocations and responses.

5.1 The Reconfiguration abstraction usage. Solid (dashed) blocks depict dynamic (resp. static) objects.

5.2 Example run of client $p_i$ of our algorithm. The dashed configuration is in $p_i$’s $ToTrack$ after calling $N_1,CoS$, and thus it is in $potentialSuccessors(N_1)$. But it is dropped after $p_i$ calls $N_2,CoS$, and thus it is never introduced, and so it is not in $successors(N_2)$.
List of Tables

2.1  **Space hierarchy**: Number of base objects for $f$-tolerant register emulation with $k$ writers and $n \geq 2f + 1$ servers. ........................ 10
Abstract

In recent years we see an exponential increase in storage capacity demands, creating a need for big data storage solutions. Additionally, today’s economy emphasizes consolidation, giving rise to massive data centers and clouds. In this era, distributed storage plays a key role. This is marked by two clear trends: First, the storage market gravitates towards distributed storage solutions within data centers as well as across multiple data centers; such systems are typically made up of many cheap, low-reliability storage nodes, and achieve durability and high availability via redundancy. Second, we see more and more users store data in clouds that are accessed remotely over the Internet. In this thesis, we study two fundamental aspects of reliable distributed storage: storage space cost and dynamic storage reconfiguration.

**Storage space cost.** Given the inherent failure-prone nature of storage and network components, a reliable distributed storage algorithm must store redundant information in order to allow data to be reconstructed when some subset of the system fails. The most common approach to achieve this is via replication, i.e., storing multiple copies of each data block. The well-known ABD result shows that an \( f \)-tolerant storage can be emulated using a collection of \( 2f + 1 \) fault-prone servers, each storing a single read-modify-write object type, which is known to be optimal. We first generalize this bound by investigating the inherent space cost of emulating reliable storage as a function of the object type exposed by the underlying servers. Then, we focus on read-modify-write object type and investigate whether erasure codes can help mitigate the significant cost of replication that results from the immense size of the data. Some previous works attempted to do it, but a closer look at existing solutions reveals that they incur extra storage costs in other places. Specifically, the use of coding creates scenarios where old information cannot be over-written, leading the storage to grow without bound if concurrency is high. We shed light on the fundamental tradeoff between lower redundancy per block (as achieved by codes), concurrency, and allowing old data to be over-written (as achieved by replication).

**Dynamic reconfiguration.** A key challenge for distributed storage is the problem of reconfiguration. Clearly, any production storage system that provides data reliability and availability for long periods must be able to reconfigure in order to remove failed
or old nodes and add healthy or new ones. Reconfiguration is also essential in order to realize the greatest advantage of distributed storage over traditional monolithic enterprise storage solutions, namely, that it supports elastic incremental growth— the main motivation for the economic model of cloud computing. In this thesis we contribute to the understanding of reconfiguration of distributed storage by showing both negative (impossibility) and positive (algorithms) results.
List of Acronyms

RMW  read modify write
MWMR multi-writer multi-reader
MWSR multi-writer single-reader
SWSR single-writer single-reader
SWMR single-writer multi-reader
RR ranked register
CoS common set
List of Symbols

\( \diamond p^D \)  dynamic failure detector
Chapter 1

Introduction

Data collection and consumption is growing exponentially in recent years (see, e.g., [46]), and is expected to continue to grow even faster [76]. This creates a demand for larger and larger storage capacities and support for elastic growth thereof. The growth of individual disk capacities cannot match this phenomenal surge in demand, and so distributed storage has become the method of choice. Storage solutions nowadays consist of large interconnected collections of disks, controllers, and servers. In some cases they reside within a single data center, while in others they are geo-distributed so as to allow local access to geographically dispersed clients as well as for disaster recovery. In many cases the storage is provided as a cloud-based service. In this thesis, we deal with fundamental challenges that arise in today’s massive distributed storage. In particular, we study reliable asynchronous storage space cost and its dynamic reconfiguration. Part of the results in this thesis were previously published in PODC [27, 69], DISC [73], and SIROCCO [71].

1.1 Storage cost

Reliable storage emulations seek to construct fault-tolerant shared objects, such as read/write registers, using a collection of base objects hosted on failure-prone servers. A reliable distributed storage emulation must therefore store redundant information in order to allow data to be reconstructed when some subset of the system fails. The most common approach to achieve this is via replication, i.e., storing multiple copies of each data block, and most existing storage emulation algorithms are constructed from storage services capable of supporting custom-built read-modify-write (RMW) primitives [10, 35, 41, 31, 39, 7, 64, 57], where perhaps the most famous one is ABD [10]. This algorithm emulates an \( f \)-tolerant atomic wait-free register, accessed by an unbounded number of processes (readers and writers), on top of \( 2f + 1 \) servers, each of which stores a single RMW object.

The power of primitives. Since \( f \)-tolerant register emulation is impossible with less than \( 2f + 1 \) servers [14, 55], the ABD algorithm’s space complexity is optimal in terms of
number of base objects. However, support for atomic RMW is not always available: the operations exposed by network-attached disks are sometimes limited to basic read/write capabilities, and the interfaces exposed by cloud storage services sometimes augment this with simple conditional update primitives similar to Compare-and-Swap (CAS). A natural question that we study in Chapter 2 is therefore how the ABD results generalize when only weaker primitives (e.g., read/write registers) are available. More specifically, we are interested whether reliable storage emulations are possible with weaker primitives, and if so, what their space cost in terms of number of objects is, and in particular, does their cost depend on the number of writers and the number of servers. This question led us to define a new space hierarchy of fault tolerant storage emulations.

**Erasure codes.** Given the immense size of data, the storage cost of full replication becomes significant. A promising approach mitigates this cost via the use of erasure codes \[6, 21, 42, 23, 77, 30\]. In a nutshell, erasure codes allow redundant information to be shared among multiple data blocks. For example, it is possible to keep a single redundant block constructed as the xor of \( k \) data blocks instead of holding replicas of all \( k \) blocks. Either way, the data can be reconstructed despite the loss of any single (original or redundant) block as illustrated in Figure 1.1. Given data of size \( D \) bits and \( f = 1 \), this approach can ideally reduce the storage cost from \( 3D \) to \((k + 1)D/k\) bits. If a large \( k \) is used, the redundancy overhead with erasure codes is diminished, and the gap becomes significant.

While this approach is appealing, a closer look at existing erasure-coded storage solutions reveals that they incur extra storage costs somewhere else. Specifically, the use of coding in lieu of replication creates a need for storing multiple versions of each data block, and the number of version can grow without bound if concurrency is high \[42, 23\]. To avoid it, some keep unbounded information in channels \[30, 21\] while others assume periods of synchrony \[77\] or allow returning obsolete values \[77\].

In Chapter 3 we study to what extent such a storage blowup is inevitable. We investigate the fundamental limitations of coded storage and the tradeoffs between replication
and coding. In particular, we first prove a lower bound on the storage cost, in terms of bits, for asynchronous algorithms under some assumptions stated in Chapter 3. Our lower bound is presented as a function of the number of storage node failures tolerated, the concurrency allowed by the algorithm, and the data size. Then, we also present in Chapter 3 an algorithm that combines erasure-codes with replication to achieve an asymptotically tight upper bound.

1.2 Dynamic reconfiguration

A second key challenge for distributed storage is reconfiguration. Clearly, any production storage system that provides data reliability and availability for long periods must be able to reconfigure in order to remove failed or old nodes and add healthy or new ones. Reconfiguration is also essential in order to realize the greatest advantage of distributed storage over traditional monolithic enterprise storage solutions, namely, that it supports elastic incremental growth— the main motivation for the economic model of cloud computing.

The study of reconfigurable replication has been active since at least the early 1980s, with the development of group communication and virtual synchrony (see survey in [28]). But despite its vital real-world importance, the issue of online reconfiguration of storage services is still not well understood; theory for this problem is in its infancy, and most distributed storage systems in production are reconfigured manually during complete system shutdown.

Until recently, no liveness (progress) properties for a reconfigurable storage service were formally specified. While it has long been well-known that, in a static system, asynchronous replication requires a majority of replicas to be operational for liveness, no similar criterion had been formulated for the case where the set of replicas dynamically changes. The DynaStore [8] algorithm was the first to formulate sufficient conditions for liveness of reconfigurable storage. Nevertheless, these conditions were never shown to be necessary. For example, DynaStore, like its predecessors [40, 41, 53, 24], ensures progress only if the number of reconfigurations is finite.

Liveness of Dynamic Storage In Chapter 4, we prove that this is indeed inherent for asynchronous API-based storage solutions, where the service’s API is extended with a reconfiguration operation for changing the current configuration of participating processes [66, 8, 50, 37, 40, 41, 24], which can be only invoked by members of the current configuration. In addition, we define in Chapter 4 a dynamic failure detector that can be easily implemented in eventually synchronous systems, and use it to implement dynamic storage algorithm that ensures liveness for all operations without restricting the reconfiguration rate.
Dynamic fault model. Moreover, DynaStore’s restrictions on failures are not defined in an intuitive way, and do not create a clear distinction between user-controlled actions (like removing a process from the current configuration) and ones stemming from the environment (like process failures). In Chapter 5 we first define a dynamic model with a clean failure condition that allows an administrator to reconfigure the system and switch off a server once the reconfiguration operation removing it completes. Our dynamic fault model is defined as an interplay between the object’s implementation and its environment.

Reconfiguration abstraction and optimal solution. DynaStore is quite complex, the reconfiguration mechanism is not clearly abstracted away from the intricacies of data replication, its operation complexity is exponential, and it does not explicitly deal with clients and only presents a server-side algorithm. (Follow-up work [66] extends DynaStore to support client operations, but this is systems work that does not deal with the topic formally). In Chapter 5 we also define a Reconfiguration abstraction in the client-server model, show how it can be used to build dynamic storage, and give an optimal asynchronous algorithm implementing the Reconfiguration abstraction, which in turn leads to the first asynchronous storage emulation with optimal linear complexity.
Chapter 2

The Power of Primitives for Fault-Tolerant Register Emulations

Reliable storage emulations seek to construct fault-tolerant shared objects, such as read/write registers, using a collection of base objects hosted on failure-prone servers. Such emulations are core enablers for many modern storage services and applications, including cloud-based online data stores [29, 61, 60, 47, 58] and Storage-as-a-Service offerings [63, 67, 33, 75].

Most existing storage emulation algorithms are constructed from storage services capable of supporting custom-built read-modify-write (RMW) primitives [10, 35, 35, 41, 31, 39, 7, 64, 57], and perhaps the most famous one is ABD [10]. This algorithm emulates an $f$-tolerant atomic wait-free register, accessed by an unbounded number of processes (readers and writers), on top of $2f+1$ servers, each of which stores a single RMW object. Since $f$-tolerant register emulation is impossible with less than $2f+1$ servers [14, 55], the ABD algorithm’s space complexity is optimal.

However, support for atomic RMW is not always available: the operations exposed by network-attached disks are typically limited to basic read/write capabilities, and the interfaces exposed by cloud storage services sometimes augment this with simple conditional update primitives similar to Compare-and-Swap (CAS). A natural question that arises is therefore how the ABD results generalize when only weaker primitives (e.g., read/write registers) are available. More specifically, we are interested whether reliable storage emulations are possible with weaker primitives, and if so, what their space complexity is, and in particular, does their complexity depend on the number of writers and the number of servers. These questions lead us to define a new space hierarchy of fault tolerant register emulations.
Chapter 2. The Power of Primitives for Fault-Tolerant Register Emulations

<table>
<thead>
<tr>
<th>Base object</th>
<th>Lower bound</th>
<th>Upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(WS-Safe, obstruction-free)</td>
<td>(WS-Regular, wait-free)</td>
</tr>
<tr>
<td>max-register</td>
<td>$2f + 1$</td>
<td>$2f + 1$</td>
</tr>
<tr>
<td>CAS</td>
<td>$2f + 1$</td>
<td>$2f + 1$</td>
</tr>
<tr>
<td>read/write register</td>
<td>$kf + \left\lceil \frac{k}{2f+1} \right\rceil (f + 1)$</td>
<td>$kf + \left\lceil \frac{k}{2f+1} \right\rceil (f + 1)$</td>
</tr>
</tbody>
</table>

Table 2.1: **Space hierarchy**: Number of base objects for $f$-tolerant register emulation with $k$ writers and $n \geq 2f + 1$ servers.

**Space hierarchy.** Herlihy’s well-known consensus hierarchy [44] provides a way to compare the power of different primitives (e.g., read/write register, T&S, CAS, etc), in terms of computability. However, this hierarchy tells us nothing about the space complexity of the corresponding emulations. A recent paper by Ellen et al. [34] sheds some light on this topic by presenting a hierarchy classified by the space complexity required for solving obstruction-free consensus. We follow Ellen et al. [34], and make another step towards understanding the space requirements induced by different primitives.

We introduce a new space hierarchy, which classifies primitives by the space required for reliable register emulations on top of fault-prone servers that support the primitives. More specifically, we assume a collection of $n$ fault-prone servers, each of which stores base objects supporting the given primitive. The failure granularity is servers, meaning that a server crash causes all base objects it stores to crash as well. We study three primitives: read/write register, max-register [9], and CAS. For each primitive, we are interested in the number of base objects required to emulate an $f$-tolerant register for $k$ writers using $n$ servers.

To strengthen our result, we prove the lower bound under weak liveness and safety guarantees, namely, obstruction freedom and write sequential safety (WS-Safety). The latter is a weak generalization for Lamport’s notion of safety [52] to multi-writer registers, which we define in Section 2.1. Since atomicity usually requires readers to write, which may induce a dependency on the number of readers, we consider regularity for our upper bound; we define in Section 2.1 write sequential regularity (WS-Regularity), which is a weaker form of multi-writer regularity defined in [64]. The lower bound proved in [14, 55] on the number of servers required for $f$-tolerant register emulations can be easily generalized for WS-Safe obstruction-free emulations. Therefore, we assume that $n \geq 2f + 1$ throughout the chapter.

Table 2.1 summarizes the hierarchy. Interestingly, even though both registers and max-registers have the lowest consensus number of 1 in Herlihy’s hierarchy [44], they have different powers in our hierarchy, whereas CAS, which has an infinite consensus number, has the same power as max-register in our hierarchy. As an aside, we note that our hierarchy has implications for the standard shared memory model (without base object failures); for example, it implies that a max-register for $k$ writers cannot be implemented from less than $k$ read/write registers (proven in Theorem 3).
**Results.** Despite the fact that the original ABD emulation [10] assumes a general RMW base object on every server, we observe that the code executed by each server in the multi-writer ABD protocol [41, 64, 57] can be encapsulated into the `write-max` (for handling update messages) and `read-max` (for handling read messages) primitives of max-register. Therefore, the upper bound of $2f + 1$ applies to max-registers as well. In Section 2.2 we show how to emulate a max-register from a single CAS in a wait-free manner. Thus, the upper bound for max-register also applies to CAS.

Our main technical contribution is a lower bound on the number of read/write registers required to emulate an $f$-tolerant WS-Safe obstruction-free register for $k$ clients using $n$ servers. While the ABD [10] space complexity does not depend on the number of writers or the number of servers, we show in Section 2.3 (Theorem 2) that when servers support only read/write registers, the lower bound increases linearly with the number of writers and decreases (up to a certain point) with the number of available servers. In particular, our lower bound implies that at least $kf + f + 1$ registers are needed regardless of the number of available servers. We exploit asynchrony to show that an emulated write must complete even if it leaves $f$ pending writes on base registers, forcing the next writer to use a different set of registers, even in a write-sequential run.

In Section 2.3, we also present a new upper bound construction that closely matches our lower bound (Theorem 4). Note that the two bounds are closely aligned, and in particular, coincide in the two important cases of $n = 2f + 1$ and $n \geq kf + f + 1$ where they are equal to $kf + k(f + 1)$ and $kf + f + 1$ respectively. An interesting open question is to close the remaining small gap.

Another open question is whether our lower bound is tight for stronger regularity definitions [65]. In the special case of $n = 2f + 1$ servers and $k$ writers, a matching upper bound of $(2f + 1)k$ registers can be achieved by simply having each server implement a single $k$-writer max-register from $k$ base registers [9]. The question of the general case of $n \geq 2f + 1$ remains open.

In Appendix .1, we show the following three additional results implied by an extended variant of our main lower bound construction: (1) a lower bound of $k$ registers per server for $n = 2f + 1$ (Theorem 13); (2) a lower bound on the number of servers when the maximum number of registers stored on each server is bounded by a known constant (Theorem 14); and (3) impossibility of constructing fault-tolerant multi-writer register emulations adaptive to point contention [2, 13] (Theorem 15).

**Related work.** The space complexity of fault-tolerant register emulations has been explored in a number of prior works. Afek et al. [3] consider a model where faulty servers are responsive but may return arbitrary values, and present several storage efficient algorithms for reliable registers. We prove that when faulty servers are unresponsive, the additional storage cost is inherent. Aguilera et al. [5] present bounds on the number of fault-prone base registers required to support a reliable multi-writer one using uniform algorithms (i.e., algorithms that do not depend on a priori knowledge of the number of writers). Our results, in contrast, do not assume uniformity, and provide bounds on the
number of base registers as a function of the number of writers, servers, and the failure threshold. In Chapter 3 we consider the space complexity of reliable register emulations in terms of the amount of data bits stored on fault-prone RMW servers, and since in this chapter we are only interested in the number of stored registers and not their sizes, these results are orthogonal.

Basescu et al. [18] describe several fault-tolerant multi-writer register emulations from a collection of fault-prone read/write data stores. Their algorithms incorporate a garbage-collection mechanism that ensures that the storage cost is adaptive to the write concurrency, provided that the underlying servers can be accessed in a synchronous fashion. Our results show that asynchrony has a profound impact on storage consumption by exhibiting a sequential failure-free run where the number of registers that need to be stored grows linearly with the number of writers.

The proof of our main result (see Lemma 2) further extends the adversarial framework we use in Chapter 3 to exploits the notion of register covering (originally due to [?]) extended to fault-prone base registers as in [5]. Covering arguments have been successfully applied to proving numerous space lower bound results in the literature (see [12] for a survey) including the recent tight bounds for obstruction-free consensus [38, 78], which are at the heart of the space hierarchy of [34] discussed above.
2.1 Model

2.1.1 Shared Objects

A shared object supports concurrent execution of operations performed by some set, \( C = \{c_1, c_2, \ldots \} \), of client processes. Each operation has an invocation and response. An object schedule is a sequence of the operation invocations and their responses. An invoked operation is complete in a given schedule if the operation’s response is also present in the schedule, and pending otherwise. For a schedule \( \sigma \), \( \text{ops}(\sigma) \) denotes the set of all operations that were invoked in \( \sigma \), and \( \text{complete}(\sigma) \) (resp., \( \text{pending}(\sigma) \)) denotes the subset of \( \text{ops}(\sigma) \) consisting of all the complete (resp., pending) operations. Also, for a set \( X \) of operations, we use \( \sigma|X \) to denote the subsequence of \( \sigma \) consisting of all the invocation and responses of the operations in \( X \).

An operation \( op \) precedes an operation \( op' \) in a schedule \( \sigma \), denoted \( op \prec_{\sigma} op' \) iff \( op' \) is invoked after \( op \) responds in \( \sigma \). Operations \( op \) and \( op' \) are concurrent in \( \sigma \), if neither one precedes the other. A schedule with no concurrent operations is sequential. Given a schedule \( \sigma \), we use \( \sigma|i \) to denote the subsequence of \( \sigma \) consisting of the actions client \( c_i \). The schedule is well-formed if each \( \sigma|i \) is a sequential schedule. In the following, we will only consider well-formed schedules.

The object’s sequential specification is a collection of the object’s sequential schedules in which all operations are complete. For an object schedule \( \sigma \), a linearization \( L_{\sigma} \) of \( \sigma \) is a sequential schedule consisting of all operations in \( \text{complete}(\sigma) \) along with their responses and a subset of \( \text{pending}(\sigma) \), each of which being assigned a matching response, so that \( L_{\sigma} \) satisfies both the \( \sigma \)’s operation precedence relation (\( \prec_{\sigma} \)) and the object sequential specification.

2.1.2 Registers

A read/write register object (or simply a register) supports two operations of the form \( \text{write}(v), \ v \in \text{Vals} \), and \( \text{read()} \) returning \( \text{ack} \) and \( v \in \text{Vals} \) respectively where \( \text{Vals} \) is the register value domain. Its sequential specification is the collection of all sequential schedules where every \( \text{read} \) returns the value written by the last preceding \( \text{write} \) or an initial value \( v_0 \in \text{Vals} \) if no such write exists.

A register is multi-writer (MW) (resp., multi-reader (MR)) if it can be written (resp., read) by an unbounded number of clients. A \( k \)-writer register, or simply, \( k \)-register, is a register that can be written by at most \( k > 0 \) distinct clients. A register is single-writer (SW) (resp., single-reader (SR)) if only one process can write (resp., read) the register. For a register schedule \( \sigma \), we use \( \text{writes}(\sigma) \) and \( \text{reads}(\sigma) \) to denote the sets of all write and read operations invoked in \( \sigma \). We say that \( \sigma \) is write-sequential if no two writes in \( \text{writes}(\sigma) \) are concurrent.
2.1.3 Consistency Conditions

Consistency conditions specify the shared object behaviour when accessed concurrently by the clients. Below, we introduce a number of consistency conditions that will be used throughout this paper. They are expressed as a set of schedules $C$ satisfying one of the following requirements:

**Atomicity** [45] For all schedules $\sigma \in C$, $\sigma$ has a linearization.

**Write-Sequential Regularity (WS-Reg)** For all $\sigma \in C$, if $\sigma$ is write-sequential, then for each $rd \in \text{reads}(\sigma) \cap \text{complete}(\sigma)$ there is a linearization $L_{rd}$ of $\sigma|\text{writes}(\sigma) \cup \{rd\}$.

**Write-Sequential Safety (WS-Safe)** As WS-Reg, but only required to hold for complete reads that are not concurrent with any writes.

2.1.4 System Model

We consider an asynchronous fault-prone shared memory system [48] consisting of a set of shared base objects $B = \{b_1, b_2, \ldots\}$. The objects are accessed by a collection of clients in the set $C = \{c_1, c_2, \ldots\}$.

We consider a slight generalization of the model in [48] where the objects are mapped to a set $S = \{s_1, s_2, \ldots\}$ of servers via a function $\delta$ from $B$ to $S$. For $B \subseteq B$, we will write $\delta(B)$ to denote the image of $B$, i.e., $\delta(B) = \{\delta(b) : b \in B\}$. Conversely, for $S \subseteq S$, we will write $\delta^{-1}(S)$ to denote the pre-image of $S$, i.e., $\delta^{-1}(S) = \{b : \delta(b) \in S\}$. Note that for all $B \subseteq B$, $|\delta(B)| \leq |B|$, and conversely, for all $S \subseteq S$, $|\delta^{-1}(S)| \geq |S|$. Both servers and clients can fail by crashing. A crash of a server causes all objects mapped to that server to instantaneously crash.

We study algorithms emulating reliable $k$-writer registers to a set of clients. Clients interact with the emulated register via high-level read and write operations. To distinguish the high-level emulated reads and writes from low-level base object invocations, we refer to the former as READ and WRITE. We say that high-level operations are invoked and return whereas low-level operations are triggered and respond. A high-level operation consists of a series of trigger and respond actions on base objects, starting with the operation’s invocation and ending with its return. Since base objects are crash-prone, we assume that the clients can trigger several operations in a row without waiting for the previously triggered operations to respond.

An emulation algorithm $A$ defines the behaviour of clients as deterministic state machines where state transitions are associated with actions, such as trigger/response of low-level operations. A configuration is a mapping to states from system components, i.e., clients and base objects. An initial configuration is one where all components are in their initial states.

\footnote{Note that the original faulty shared model of [48] can be derived from our model by choosing $\delta$ to be an injective function.}
A run of $A$ is a (finite or infinite) sequence of alternating configurations and actions, beginning with some initial configuration, such that configuration transitions occur according to $A$. We use the notion of time $t$ during a run $r$ to refer to the configuration reached after the $t$th action in $r$. A run fragment is a contiguous sub-sequence of a run. A run is write-only if it has no invocations of the high-level READ operations.

We say that a base object, client, or server is faulty in a run $r$ if it fails at some time in $r$, and correct, otherwise. A run is fair if (1) for every low-level operation triggered by a correct client on a correct base object, there is eventually a matching response, and (2) every correct client gets infinitely many opportunities to both trigger a low-level operation and execute the return actions. We say that a low-level operation on a base object is pending in run $r$ if it was triggered but has no matching response in $r$. We assume that the base objects are atomic (as defined in Section 2.1.3).

2.1.5 Properties of the Emulation Algorithms

**Safety** The emulation algorithm safety will be expressed in terms of the consistency conditions specified in Section 2.1.3. An emulation algorithm $A$ satisfies a consistency condition $C$ if for all $A$’s runs $r$, the subsequence of $r$ consisting of the invocations and responses of the high-level read and write operations is a schedule in $C$.

**Liveness** We consider the following liveness conditions that must be satisfied in fair runs of an emulation algorithm. A wait-free object is one that guarantees that every high-level operation invoked by a correct client eventually returns, regardless of the actions of other clients. An obstruction-free object guarantees that every high-level operation invoked by a correct client that is not concurrent to any other operation by a correct client eventually returns.

**Fault-Tolerance** The emulation algorithm is $f$-tolerant if it remains correct (in the sense of its safety and liveness properties) as long as at most $f$ servers crash for a fixed $f > 0$.

**Complexity measures** The resource consumption of an emulation algorithm $A$ in a (finite) run $r$ is the number of base objects used by $A$ in $r$. The resource complexity [48] of $A$ is the maximum resource consumption of $A$ in all its runs.

2.2 A Max-Register Emulation With One CAS

We present here a wait-free emulation of an atomic max-register on top of a single CAS object. The pseudocode appears in Algorithm 1. The CAS object supports one operation with two parameters, $exp$ and $new$; if $exp$ is equal to the object’s current value, then the value is set to $new$. In any case, the operation returns the old value.

A max-register supports two operations, $\text{max-write}(v)$ for some $v$ from some domain of ordered values $V$ that returns ok, and $\text{max-read}()$ that returns a value from $V$. The sequential specification of max-register is the following: A $\text{max-read}$ returns the highest value among those written by $\text{max-write}$ before it, or $v_0$ in case no such values.
Algorithm 1 Max-register emulation from a single CAS object $b$

Local variables:
- $tmp \in \mathbb{V}$, initially $v_0$
- operation $b$.CAS($exp, new$), $exp, new \in \mathbb{V}$
  - $prev \leftarrow b$
  - if $exp = b$ then
    - $b \leftarrow new$
  - return $prev$

1: operation Max-write($v$)
2: while true do
3: $tmp \leftarrow b$.CAS($v_0, v_0$)
4: if $tmp \geq v$ then
5: return ok
6: $b$.CAS($tmp, v$)
7: operation Max-read()
8: $tmp \leftarrow b$.CAS($v_0, v_0$)
9: return $tmp$

2.2.1 Correctness

We say that a $b$.CAS($exp, new$) operation is successful if $b$ is set to $new$.

Observation 1. Consider a successful $b$.CAS($exp, new$) operation for some $exp$ and $new$, the the next $b$.CAS($exp', new'$) operation for some $exp'$ and $new'$ returns $new$.

The following observation follows immediately from the fact that $b$.CAS($exp, new$) is called only with $new \geq exp$.

Observation 2. The values returned by $b$.CAS($exp, new$) are monotonically increasing.

We next define linearization points:

Definition 1. (linearization points)

max-read: The linearization point is line 8.

max-write: If the operation performs a successful $b$.CAS($tmp, v$) in line 6, then the linearization point is the last time Line 6 is performed. Otherwise, the linearization point is last time line 3 is performed.

Lemma 1. For every run $r$ of Algorithm 1, the sequential run $\sigma_r$, produced by the linearization points of operations in $r$, is a linearization of $r$.

Proof. The real time order of $r$ is trivially preserved in $\sigma_r$. We need to show that $\sigma_r$ preserves max-register’s sequential specification. Let $mr$ be a max-read() in $r$ that returns a value $v$, and let $t_{mr}$ be the time when $b$.CAS($v_0, v_0$) (that returns $v$) is called by $mr$ (line 8). We need to show that (1) there is no max-write($v'$) that precedes $mr$ in $\sigma_r$ s.t. $v' > v$, and (2) if $v \neq v_0$, there is a max-write($v$) that precedes $mr$ in $\sigma_r$. 

1. Assume by way of contradiction that there is a max-write\((v')\) \(w\) that precedes \(mr\) in \(\sigma_r\) s.t. \(v' > v\). Denote \(t_w\) to be the linearization point of \(w\) in \(r\), and note that \(t_w < t_{mr}\). Now consider two case:

(a) \(w\) performs line 6 at time \(t_w\) (line 6 is \(w\)'s linearization point). In this case, \(w\) performs a successful \(b.CAS(tmp, v')\) in line 6 for some \(tmp\) at some time \(t' \leq t_w < t_{mr}\). Therefore, by Observations 1 and 2, \(b.CAS(v_0, v_0)\) called by \(mr\) in line 8 at time \(t_{mr}\) returns \(v'' \geq v' > v\). Hence, \(mr\) returns \(v'' \geq v' > v\). A contradiction.

(b) \(w\) performs line 3 at time \(t_w\) (line 3 is \(w\)'s linearization point). Thus, \(t_w\) is the last time \(w\) calls \(tmp \leftarrow b.CAS(v_0, v_0)\) in line 3. Therefore, \(b.CAS(v_0, v_0)\), called at time \(t_w\) returns a value bigger than or equal to \(v'\). Therefore, by Observations 1 and 2, \(b.CAS(v_0, v_0)\) called by \(mr\) in line 8 at time \(t_{mr}\) returns \(v'' \geq v' > v\). Hence, \(mr\) returns \(v'' \geq v' > v\). A contradiction.

2. Assume that \(v \neq v_0\). By the code and by the CAS properties, there is a max-write\((v)\) operation \(w\) that calls a successful \(b.CAS(tmp, v)\) (line 6) for some \(tmp\) at time \(t_w < t_{mr}\). Therefore, by Observations 1 and 2, the next call to \(b.CAS(v_0, v_0)\) (in line 3) by \(w\) returns \(tmp \geq v\), and thus \(w\) does not perform line 6 again. We get that \(t_w\) is the linearization point of \(w\), and thus \(w\) precedes \(mr\) in \(\sigma_r\).

\[\square\]

**Theorem 1.** Algorithm 1 emulates wait-free atomic max-register.

**Proof.** Atomicity follows from Lemma 1. We left to show wait-freedom:

- **Max-read()**: Since \(b.CAS(exp, new)\) is wait free, Max-read() is wait-free.

- **Max-write\((v)\)**: Note that Max-write\((v)\) returns in the iteration in which \(b.CAS(v_0, v_0)\) in line 3 returns a value bigger than or equal to \(v\). Therefore, by Observations 1 and 2, Max-write\((v)\) returns in the following iteration after a successful \(b.CAS(tmp, v)\) in line 6. Now assume in a way of contradiction that Max-write\((v)\) do not have a successful \(b.CAS(tmp, v)\) in line 6. By the code and Observations 1 and 2, if \(b.CAS(tmp, v)\) in line 6 do not succeed, then the following \(b.CAS(v_0, v_0)\) in line 3 returns a bigger value that what it returned in the previous iteration. Now let \(v'\) be the value returned by the first \(b.CAS(v_0, v_0)\) in line 3, and assume w.l.o.g. that there are \(k\) values bigger than \(v'\) and smaller than \(v\) in \(V\). Therefore, after at most \(k\) iteration Max-write\((v)\) returns.

\[\square\]
2.3 Resource Complexity of Write-Sequential k-register Emulation

In Section 2.3.1 we give an overview and intuition for our lower bound, and in Section 2.3.2 we prove it. In Section 2.3.3 we present a closely matching upper bound algorithm.

2.3.1 Lower bound overview

We prove that any \( f \)-tolerant emulation of an obstruction-free WS-Safe k-register from a collection of MWMR atomic registers stored on a collection \( S \) of crash-prone servers has resource complexity of at least \( k f + \left\lceil \frac{k f}{|S| - (f + 1)} \right\rceil (f + 1) \).

Our proof exploits the fact that the environment is allowed to prevent a pending low-level write from taking effect for arbitrarily long [5]. As a result, a client executing a high-level WRITE operation cannot reliably store the requested value in a base register that has a pending write as this write may take effect at a later time thus erasing the stored value. At the same time, the client cannot wait for all base registers on which it triggers low-level operations to respond, since up to \( f \) of them may reside on faulty servers. It therefore must be able to complete a high-level write without waiting for responses from up to \( f \) registers. Consequently, the next high-level write (by a different client) cannot reliably use these registers (as they might have outstanding low-level writes), and is therefore forced to use additional registers thus causing the total number of registers to grow with each subsequent write.

In our main lemma (Lemma 2), we formalize this intuition as follows: Starting from a run \( r_0 \) consisting of an initial configuration, we build a sequence of consecutive extensions \( r_1, \ldots, r_k \) so that \( r_i \) is obtained from \( r_{i-1} \) by having a new client invoke a high-level write \( W_i \) of some (not previously used) value. We then let the environment behave in an adversarial fashion (Definition 4) by blocking the responses from the writes triggered on at most \( f \) base registers as well as the prior pending low-level writes. In Lemma 4, we show that \( W_i \) must terminate without waiting for these responses to arrive. Furthermore, in Lemma 5, we show that \( W_i \) must invoke low-level writes on at least \( 2f + 1 \) base registers (residing on \( \geq 2f + 1 \) different servers) that do not have any prior pending writes. This, combined with Lemma 4, implies that by the time \( W_i \) completes, there are at least \( f \) more registers on at least \( f \) servers with pending writes after \( W_i \) completes. Thus, by the time the \( k \)th high-level write completes, the total number of covered registers is at least \( kf \) (see Lemma 2(a)).

To obtain a stronger bound, our construction is parameterized by an arbitrary subset \( F \) of servers such that \( |F| = f + 1 \). We show that the extra storage available on these servers cannot in fact be utilized by an emulation (see Lemma 2(b)) forcing it to use at least \( kf \) registers on the remaining \( S \setminus F \) servers to accommodate the same number \( k \) of writers. We use this result in Theorem 2, to show that the number of base registers
2.3. Resource Complexity of Write-Sequential k-register Emulation

required for the emulation is a function of $k$ and $|S|$.

2.3.2 Lower Bounds

For any time $t$ (following the $t^{th}$ action) in a run $r$ of the emulation algorithm we define the following:

- **Covering write**: Let $w$ be a low-level write triggered on a base register $b$ at times $\leq t$. We will refer to $w$ as covering at $t$, and to $b$ as being covered by $w$ at $t$ if $w$ is pending at $t$.

- $C(t) \subseteq C$: the set of all clients that have completed a high-level write operation at times $\leq t$.

- $\text{Cov}(t) \subseteq B$: the set of all base registers being covered by some low-level write at time $t$.

We first prove the following key lemma:

**Lemma 2.** For all $k > 0$, $f > 0$, let $A$ be an $f$-tolerant algorithm that emulates a WS-Safe obstruction-free $k$-register using a collection $S$ of servers storing a collection $B$ of wait-free MWMR atomic registers. Then, for every $F \subseteq S$ such that $|F| = f + 1$, there exist $k$ failure-free runs $r_i$, $0 \leq i \leq k$, of $A$ such that (1) $r_0$ is a run consisting of an initial configuration and $t_0 = 0$ steps, and (2) for all $i \in [k]$, $r_i$ is a write-only sequential extension of $r_{i-1}$ ending at time $t_i > 0$ that consists of $i$ complete high-level writes of $i$ distinct values $v_1, \ldots, v_i$ by $i$ distinct clients $c_1, \ldots, c_i$ such that:

- $|\text{Cov}(t_i)| \geq i f$
- $\delta(\text{Cov}(t_i)) \cap F = \emptyset$

Fix arbitrary $k > 0$, $f > 0$, and a set $F$ of servers such that $|F| = f + 1$. We proceed by induction on $i$, $0 \leq i \leq k$. **Base**: Trivially holds for the run $r_0$ of $A$ consisting of $t_0 = 0$ steps. **Step**: Assume that $r_{i-1}$ exists for all $i \in [k - 1]$. We show how $r_{i-1}$ can be extended up to time $t_i > t_{i-1}$ so that the lemma holds for the resulting run $r_i$. For the remainder of the Lemma 2’s proof, we will assume without loss of generality that every low-level write operation is linearized simultaneously with its respond step. Formally:

**Assumption 1** (Write Linearization). For every extension $r$ of $r_{i-1}$ and a base object $b \in B$, let $L_r|b$ be a linearization of $r|b$. Then, $L_r|b$ does not include any low-level write operations in $\text{pending}(r|b)$, and for any two low-level writes $w_1, w_2 \in \text{complete}(r|b)$ such that $\text{respond}(w_1) \prec_r w_2$, $w_1 \prec_{L_r|b} w_2$.

Note that the above implies that no low-level write $w$ that is covering some register $b$ at a time $t$ in $r$ will be observed by any low-level reads from $b$ as having taken effect until after $w$’s respond event occurs.

We proceed by introducing the following notation:
Definition 2. Let $r$ be an extension of $r_{i-1}$. For all times $t \geq t_{i-1}$ in $r$, let

1. $Tr_i(t)$: the set of all base registers which have a low-level write triggered on between $t_{i-1}$ and $t$.
2. $Rr_i(t) \subseteq Tr_i(t)$ be the set of all base registers which had a low-level write triggered on and responded (took effect) between $t_{i-1}$ and $t$.
3. $Cov_i(t) = Cov_i(t) \setminus Cov_i(t_{i-1})$ be the set of all base registers that have been newly covered between $t_{i-1}$ and $t$. Note that $Cov_i(t) \subseteq Tr_i(t)$.
4. $Q_i(t) \subseteq S$ be the set of all servers such that $Q_i(t) = \delta(Cov_i(t)) \setminus F$ if $|\delta(Cov_i(t)) \setminus F| \leq f$, and $Q_i(t) = Q_i(t-1)$, otherwise. Intuitively, $Q_i(t)$ is a set of at most $f$ servers, each of which is not in $F$ and has at least one newly covered registers (between $t_{i-1}$ and $t$).
5. $F_i(t) \triangleq \{s \in F \mid \delta^{-1}(\{s\}) \cap Rr_i(t) \neq \emptyset\}$, i.e., $F_i(t)$ is the set of all servers in $F$ having a register that responded to a low-level write invoked after $t_{i-1}$.
6. $M_i(t) \triangleq \delta(Cov_i(t)) \setminus (F \setminus F_i(t))$, i.e., $M_i(t)$ is the set of all servers in $F$ with at least one register covered by a low-level write invoked after $t_{i-1}$ and without registers that have responded to the low-level writes invoked after $t_{i-1}$.
7. $G_i(t) \subseteq S$ be the set of all servers such that $G_i(t) = M_i(t)$ if $|Q_i(t)| < |F_i(t)|$, and $G_i(t) = \emptyset$, otherwise.

Below we will introduce the adversary $Ad_i$, which causes $A$ to gradually increase the number of base registers covered after $t_{i-1}$ by delaying the respond actions of some of the previously triggered low-level writes.

Definition 3 (Blocked Writes). Let $r$ be an extension of $r_{i-1}$. For all times $t \geq t_{i-1}$ in $r$, let $BlockedWrites_i(t)\subseteq r$ be the set of all low-level covering writes $w$ satisfying either one of the following two conditions:

1. $w$ was triggered by a client in $C(t_{i-1})$, or
2. $w$ was triggered on a base register in $\delta^{-1}(Q_i(t) \cup G_i(t))$.

We say that a pending low-level write $w$ is blocked in an extension $r$ of $r_{i-1}$ if there exists a time $t \geq t_{i-1}$ such that for all $t' > t$ in $r$, $w \in BlockedWrites_i(t')$. The following definition specifies the set of the environment behaviours that are allowed by $Ad_i$ in all extensions of $r_{i-1}$:

Definition 4 ($Ad_i$). For an extension $r$ of $r_{i-1}$ we say that the environment behaves like $Ad_i$ after $r_{i-1}$ in $r$ if the following holds:

1. There are no failures after time $t_{i-1}$ in $r$. 

2. For all $t \geq t_{i-1}$ in $r$, if a low-level write $w \in \text{BlockedWrites}_i(t)$, then $w$ does not respond at $t$.

3. If $r$ is infinite then:

   (a) Every pending low-level read or write that is not blocked in $r$ eventually responds.

   (b) Every trigger or return action that is ready to be executed at a client $c$ in $r$ is eventually executed.

For a finite extension $r$ of $r_{i-1}$, we will write $\langle r, Ad_i \rangle$ to denote the set of all extensions of $r$ in which the environment behaves like $Ad_i$ after $r_{i-1}$; and we will write $\langle r, Ad_i, t \rangle$ to denote the subset of $\langle r, Ad_i \rangle$ consisting of all runs having exactly $t$ steps. For $X \in \{Q_i, F_i, M_i\}$ and a run $r \in \langle r_{i-1}, Ad_i, t \rangle$, we say that $X$ is stable after $r$ if for all $t' \geq t$ for all extensions $r' \in \langle r, Ad_i, t' \rangle$, $X(t') = X(t)$.

The following lemma asserts several technical facts implied directly by Definitions 2 and 4.

**Lemma 3.** For all $t \geq t_{i-1}$ and $r \in \langle r_{i-1}, Ad_i, t \rangle$, all of the following holds at time $t$ in $r$:

1. $Q_i(t) \subseteq \delta(Cov_i(t)) \setminus F$
2. $Q_i(t) \subseteq Q_i(t + 1)$
3. $F_i(t) \subseteq F_i(t + 1)$
4. $|F_i(t)| - |Q_i(t)| \leq 1$
5. $|Q_i(t)| \leq f$
6. $|F_i(t)| \leq f + 1$
7. $F_i(t) = F_i(t + 1) \implies M_i(t) \subseteq M_i(t + 1)$
8. $|M_i(t)| \leq f + 1$
9. $|\delta(Cov_i(t)) \setminus F| \geq f \implies |Q_i(t)| \geq f$
10. $|\delta(Cov_i(t)) \setminus F| < f \implies \delta(Rr_i(t)) \setminus F = \emptyset$
11. $(Q_i(t) \cup M_i(t)) \cap \delta^{-1}(Rr_i(t)) = \emptyset$

**Proof.** By induction on $t \geq t_{i-1}$.

**Base:** If $t = t_{i-1}$, then $Tr_i(t) = Rr_i(t) = Cov_i(t) = F_i(t) = \emptyset$. Furthermore, since $|\delta(Cov_i(t)) \setminus F| = 0 \leq 1 < f$, $Q_i(t) = \delta(Cov_i(t)) \setminus F$. Thus, all the claims hold.

**Induction step:** Suppose all the claims hold for all $t \geq t_{i-1}$, and consider the time $t + 1$:

3.1: If $|\delta(Cov_i(t + 1)) \setminus F| \leq f$, then by Definition 2.4, $Q_i(t + 1) = \delta(Cov_i(t + 1)) \setminus F$ as needed. Otherwise, $Q_i(t + 1) = Q_i(t)$. Consider an arbitrary server $s \in Q_i(t + 1)$, and towards a contradiction, suppose that $s \notin \delta(Cov_i(t + 1)) \setminus F$. Since $s \in Q_i(t + 1) = Q_i(t)$, by the induction hypothesis, $s \notin \delta(Cov_i(t)) \setminus F$. Since $s \notin \delta(Cov_i(t + 1)) \setminus F$, we get that either (1) $s \notin \delta(Cov_i(t + 1))$ or (2) $s \in \delta(Cov_i(t + 1)) \cap F$. Since $s \notin F$, (2) is false, and therefore, $s \notin \delta(Cov_i(t + 1))$. Hence, there exists a base register in $\delta^{-1}(\{s\})$ that responded at $t$ to a low-level write $w$ triggered after $t_{i-1}$. Since $s \in Q_i(t)$, by Definition 3,
Therefore, by Definition 2.3, there exists a base register $Ad_i$. This means that either no low-level writes have been triggered on registers in $b$, or there is a register $s$ that responded at time $t$ to a low-level write $w$ invoked after $t_{i-1}$. Since $s \in Q_i$, by Definition 3, $w$ is blocked writes. However, since the environment behaves like $Ad_i$ after $r_{i-1}$, by Definition 4.2, $w$ does not respond at $t$. A contradiction.

3.2: Towards a contradiction, suppose that there exists $s \in \mathcal{S}$ such that $s \in Q_i \land s \notin Q_i(t+1)$. By Definition 2.4, $|\delta(Cov_i(t+1)) \setminus F| \leq f$ as otherwise, $Q_i(t+1) = Q_i(t)$ contradicting the assumption. Thus, $Q_i(t+1) = \delta(Cov_i(t+1)) \setminus F$, and therefore, either (1) $s \notin \delta(Cov_i(t+1))$ or (2) $s \in \delta(Cov_i(t+1)) \cap F$. By the induction hypothesis for 3.1, $s \notin \delta(Cov_i(t)) \cap s \notin F$. Hence, (2) is false, and it is only left to consider the case $s \notin \delta(Cov_i(t))$. Thus, $s \in \delta(Cov_i(t))$ and $s \notin \delta(Cov_i(t+1))$, which implies that there exists a base register in $\delta^{-1}(\{s\})$ that responded at time $t$ to a low-level write $w$ invoked after $t_{i-1}$. Since $s \in Q_i$, by Definition 3, $w$ is blocked writes. However, since the environment behaves like $Ad_i$ after $r_{i-1}$, by Definition 4.2, $w$ does not respond at $t$. A contradiction.

3.3: Let $s \in F_i(t)$. By Definition 2.5, there exists a base register $b \in \delta^{-1}(\{s\})$ that responded to a low-level write triggered on $b$ after $t_{i-1}$ at time $t'$ such that $t_{i-1} < t' \leq t + 1$. Since $t < t + 1$, the $b$’s response has also occurred before $t + 1$, and therefore, $b \in R_{i-1}(t + 1)$. Hence, $b \in \delta^{-1}(\mathcal{S}) \cap R_i(t + 1) = F_i(t + 1)$ as needed.

3.4: Toward a contradiction, suppose that $|F_i(t + 1) - Q_i(t + 1)| > 1$. Since we already proved that $Q_i(t) \subseteq Q_i(t + 1) \subseteq Q_i(t + 1) - |Q_i(t)| \leq |Q_i(t + 1)|$. In addition, we know that $|Q_i(t + 1)| - |Q_i(t)| \leq 1$, and by the induction hypothesis $|F_i(t)| - |Q_i(t)| \leq 1$. Thus, $|F_i(t + 1)| - |Q_i(t + 1)| > 1$ implies that (1) $F_i(t) > Q_i(t)$, (2) $|Q_i(t + 1)| = |Q_i(t)|$, and (3) $|F_i(t + 1)| = |F_i(t)| + 1$. Since we already proved that $F_i(t + 1) \subseteq F_i(t)$, (3) implies that there exists $s \in \mathcal{S}$ such that $s \in F_i(t + 1) \setminus F_i(t)$. Since by Definition 2.5, $F_i(t + 1) \subseteq F_i(t)$. Thus, $s \in F \setminus F_i(t)$. This means that either no low-level writes have been triggered on registers in $\delta^{-1}(\{s\})$ after $t_{i-1}$, or there is a register $b \in \delta^{-1}(\{s\})$ that responds to a low-level write triggered on $b$ after $t_{i-1}$. In the first case, no register on $s$ can respond at time $t$, and therefore, $s \notin F_i(t + 1)$, which is a contradiction. In the second case, we observe that $b$ satisfies $b \in Cov_i(t)$, $b \in \delta^{-1}(\{s\})$, $s \in F \setminus F_i(t)$, and $b$ responds at $t$ to a covering write $w$ triggered after $t_{i-1}$. Thus, by Definition 2.6, $s \in M_i(t)$ and since $|F_i(t)| > |Q_i(t)|$, by Definition 2.7, $s \in G_i(t)$. Thus, by Definition 3, $w \in \mathcal{B}_i(t)$, and since the environment behaves like $Ad_i$ after $r_{i-1}$, by Definition 4.2, $w$ does not respond at $t$. A contradiction.

3.5: Assume by contradiction that $|Q_i(t + 1)| > f$. Since by 3.1, $|Q_i(t + 1)| \subseteq \delta(Cov_i(t + 1)) \setminus F$, $|\delta(Cov_i(t + 1)) \setminus F| > f$. By Definition 2.4, $Q_i(t+1) = Q_i(t)$, and therefore, $|Q_i(t)| > f$. A contradiction to the inductive assumption.

3.6: By Definition 2.5, $F_i(t + 1) \subseteq F$. Since $F = f + 1$, $|F_i(t + 1)| \leq f + 1$.

3.7: Suppose $F_i(t) = F_i(t + 1)$. Consider $s \in M_i(t)$, and toward a contradiction, suppose that $s \notin M_i(t + 1)$. Since by Definition 2.5, $F_i(t) \subseteq F$ and $F(t + 1) \subseteq F$, $F \setminus F_i(t) = F \setminus F_i(t + 1)$. This together with the fact that $s \in F \setminus F_i(t)$ implies that $s \in F \setminus F_i(t + 1)$ as well. Thus, it must be the case that $s \in \delta(Cov_i(t)) \setminus s \notin \delta(Cov_i(t + 1))$. Thus, by Definition 2.3, there exists a base register $b \in \delta^{-1}(\{s\})$ that responds to a low-
level write invoked after \( t_{i-1} \) at time \( t \) which implies that \( b \in Rr_i(t+1) \). Hence, by Definition 5, \( s \in F_i(t+1) \). However, since \( s \in F \setminus F_i(t+1), s \not\in F_i(t+1) \). A contradiction.

3.8: Since \( F \) is fixed in advance and \( |F| = f + 1 \), we receive \( |M_i(t+1)| = |\delta(Cov_i(t+1)) \cap (F \setminus F_i(t+1))| \leq |F \setminus F_i(t+1)| \leq |F| = f + 1 \).

3.9: If \( |\delta(Cov_i(t)) \setminus F| < f \) and \( |\delta(Cov_i(t+1)) \setminus F| \geq f \), then there exists a base register on a server in \( S \setminus F \) that is newly covered after \( t \). Thus, we get \( |\delta(Cov_i(t)) \setminus F| = f - 1 \) and \( |\delta(Cov_i(t+1)) \setminus F| = f \). By Definition 2.4, \( Q_i(t+1) = \delta(Cov_i(t+1)) \setminus F, \) and therefore, \( |Q_i(t+1)| = f \). Otherwise, by the induction hypothesis, \( |Q_i(t)| \geq f \). Since \( |\delta(Cov_i(t+1)) \setminus F| \geq f \), we have that either (1) \( |\delta(Cov_i(t+1)) \setminus F| = f \), or (2) \( |\delta(Cov_i(t+1)) \setminus F| > f \). Applying Definition 2.4 to (1) and (2), we get the following: for (1), \( Q_i(t+1) = \delta(Cov_i(t+1)) \setminus F \), which implies \( |Q_i(t+1)| = |\delta(Cov_i(t+1)) \setminus F| = f \), and for (2), \( Q_i(t+1) = Q_i(t) \), and therefore, \( |Q_i(t+1)| \geq f \).

3.10: Toward a contradiction, suppose that \( |\delta(Cov_i(t+1)) \setminus F| < f \) and \( |\delta(Rr_i(t+1)) \setminus F| \neq \emptyset \). By the induction hypothesis, \( |\delta(Cov_i(t)) \setminus F| < f \implies |\delta(Rr_i(t)) \setminus F| = \emptyset \). We first consider the case \( |\delta(Cov_i(t)) \setminus F| \geq f \). Thus given \( |\delta(Cov_i(t+1)) \setminus F| < f \), there exists a server in \( \delta(Cov_i(t)) \setminus F \) such that some register on that server responds to a low-level write \( w \) that was triggered after \( t_{i-1} \). Moreover, \( |\delta(Cov_i(t)) \setminus F| = f \), and thus, by Definition 2.4, \( Q_i(t) = \delta(Cov_i(t)) \setminus F \). Since the environment behaves like \( A_{d_i} \) after \( r_i \), by Definition 3, \( w \in BlockedWrites(t) \), and therefore, by Definition 4.2, \( w \) does not respond at \( t \). A contradiction.

Thus, we know that \( |\delta(Cov_i(t)) \setminus F| < f \) and \( |\delta(Rr_i(t)) \setminus F| = \emptyset \). And since \( \delta(Rr_i(t+1)) \setminus F \neq \emptyset \), there exists a server \( s \in \delta(Cov_i(t)) \setminus F \) such that some object on \( s \) responded at \( t \) to a low-level write \( w \) triggered after \( t_{i-1} \). Since \( |\delta(Cov_i(t)) \setminus F| < f \), by Definition 2.4, \( Q_i(t) = \delta(Cov_i(t)) \setminus F \). Thus, \( s \in Q_i(t) \). However, by Definition 3, \( w \in BlockedWrites(t) \), and since the environment behaves like \( A_{d_i} \) after \( t_{i-1} \), by Definition 4.2, \( w \) does not respond at \( t \). A contradiction.

3.11: Toward a contradiction, suppose that \( (Q_i(t+1) \cup M_i(t+1)) \cap \delta(Rr_i(t+1)) \neq \emptyset \). We will consider the following two cases separately: (1) \( Q_i(t+1) \cap \delta(Rr_i(t+1)) \neq \emptyset \), and (2) \( M_i(t+1) \cap \delta(Rr_i(t+1)) \neq \emptyset \).

(1) Suppose \( Q_i(t+1) \cap \delta(Rr_i(t+1)) \neq \emptyset \), and let \( s \in Q_i(t+1) \cap \delta(Rr_i(t+1)) \). If \( s \in Q_i(t) \), then by the induction hypothesis \( s \not\in \delta(Rr_i(t)) \). This means that either (a) \( \delta^{-1}(\{s\}) \cap Tr(t) = \emptyset \), or (b) there exists a base register in \( \delta^{-1}(\{s\}) \) that responds to a low-level write \( w \) triggered after \( t_{i-1} \). If (a) holds, then no base register can respond to a low-level write before \( t+1 \), and therefore, \( s \not\in \delta(Rr_i(t+1)) \), which is a contradiction. If (b) is the case, then since \( s \in Q_i(t) \), by Definition 3, \( w \in BlockedWrites(t) \). Since the environment behaves like \( A_{d_i} \) after \( t_{i-1} \), by Definition 4.2, \( w \) does not respond at \( t \), which is also a contradiction.

If \( s \not\in Q_i(t) \), then since \( s \in Q_i(t+1) \), by Definition 2.4, the only action that can follow \( t \) is a trigger of a low-level write on some object in \( \delta^{-1}(\{s\}) \). Since \( s \in \delta(Rr_i(t+1)) \), and the action executed at \( t \) is not a respond, \( s \not\in \delta(Rr_i(t)) \). Thus, \( Q_i(t+1) \neq Q_i(t) \), which by Definition 2.4, implies that \( Q_i(t+1) = \delta(Cov_i(t+1)) \setminus F \), and \( |Q_i(t+1)| = |\delta(Cov_i(t+1)) \setminus F| = f \).
\(|\delta(\text{Cov}_i(t + 1)) \setminus F| \leq f\). Since \(Q_i(t) \subset Q_i(t + 1), |Q_i(t)| < f\), and by the induction hypothesis for 3.9, we have \(|\delta(\text{Cov}_i(t + 1)) \setminus F| < f\). Thus, by the induction hypothesis for 3.10, we conclude that no registers on servers in \(S \setminus F\) have responded to any low-level writes triggered between \(t_{i-1}\) and \(t\). However, since \(s \in \delta(R_{Ri}(t))\) and, by the induction hypothesis for 3.1, \(s \notin \delta(R_{Ri}(t))\), \(s \in S \setminus F\) has a register that responded to a low-level write triggered after \(t_{i-1}\). A contradiction.

(2) Suppose that \(M_i(t + 1) \cap \delta(R_{Ri}(t + 1)) \neq \emptyset\), and let \(s \in M_i(t + 1) \cap \delta(R_{Ri}(t + 1))\). By Definition 2.5, we know that \(s \in F \setminus F_i(t + 1)\), and therefore, \(s \in F\) and \(s \notin F_i(t + 1)\). Thus, \(s \in F \cap \delta(R_{Ri}(t + 1))\), and therefore, by Definition 2.5, \(s \in F_i(t + 1)\). A contradiction. □

The following corollary follows immediately from the claims 2–3 and 5–8 of Lemma 3.

**Corollary 1.** There exists a run \(r \in \langle r_{i-1}, Ad_i \rangle\) such that \(F_i, Q_i,\) and \(M_i\) are all stable after \(r\).

We first show that \(r_{i-1}\) can be extended with a complete high-level write \(W_i\) by a new client \(c_i\) such that the environment behaves like \(Ad_i\) until \(W_i\) returns. Roughly, the reason for this is that \(Ad_i\) guarantees that after \(r_{i-1}\), \(c_i\) would only miss responses from the writes invoked at most \(f\) servers (see Claim 1) as well as those that might have been invoked in \(r_{i-1}\) by other clients \(\{c_1, \ldots, c_{i-1}\}\), which \(c_i\) is unaware of. As a result, the involved servers and clients would appear to \(c_i\) as faulty after \(r_{i-1}\), and therefore, to ensure obstruction freedom, it must complete \(W_i\) without waiting for the outstanding writes to respond.

**Lemma 4.** Let \(r \in \langle r_{i-1}, Ad_i \rangle\) be a run consisting of \(r_{i-1}\) followed by a high-level write invocation \(W_i\) by client \(c_i \notin C(t_{i-1})\). Then, there exists a run \(r' \in \langle r, Ad_i \rangle\) in which \(W_i\) returns.

By Corollary 1, there exists an extension \(r' \in \langle r, Ad_i, t' \rangle\) where \(t' > t_{i-1}\) such that \(Q_i, F_i,\) and \(M_i\) are all stable after \(r'\). If \(W_i\) returns in \(r'\), we are done. Otherwise, we will first bound the number of servers \(|Q_i(t') \cup M_i(t')|\) controlled by \(Ad_i\) as follows:

**Claim 1.** Consider a time \(t > t_{i-1}\), and a run \(r \in \langle r_{i-1}, Ad_i, t \rangle\). If \(M_i\) is stable after \(r\), then \(|Q_i(t) \cup M_i(t)| \leq f\).

**Proof.** By Lemma 3.5, \(|Q_i(t)| \leq f\). Thus, if \(M_i(t) = \emptyset\), then \(|Q_i(t) \cup M_i(t)| \leq f\) as needed. Otherwise (\(M_i(t) \neq \emptyset\)), we show that \(|F_i(t)| = |Q_i(t)| + 1\). Suppose to the contrary that \(|F_i(t)| \neq |Q_i(t)| + 1\). Since by Lemma 3.4, \(|F_i(t)| \leq |Q_i(t)| + 1\), the only possibility for \(|F_i(t)| \neq |Q_i(t)| + 1\) is if \(|F_i(t)| \leq |Q_i(t)|\). Thus, by Definition 2.7, \(G_i(t) = \emptyset\), and hence, by Definition 3, no writes on the registers in \(\delta^{-1}(M_i(t))\) are blocked. However, since \(M_i(t) \neq \emptyset\), at least one register in \(\delta^{-1}(M_i(t))\) must have an outstanding write. Therefore, by Definition 4.3(a), there exists time \(t'\), and an extension \(r' \in \langle r, Ad_i, t' \rangle\) such that one of the registers on some server \(s \in M_i(t)\) responds at time \(t'\). Thus, \(s \in F_i(t')\), and therefore, \(s \notin M_i(t')\). Hence, \(M_i\) is not stable after \(r\). A contradiction to the assumption.
Since $F_i(t) \subseteq F$, $|F \setminus F_i(t)| + |F_i(t)| = |F| = f + 1$. Hence, $|F \setminus F_i(t)| = f + 1 - |F_i(t)| = f - |Q_i(t)|$. Since by Definition 2.6, $M_i(t) \subseteq (F \setminus F_i(t))$, $|M_i(t)| \leq |F \setminus F_i(t)| = f - |Q_i(t)|$. Thus, we receive $|Q_i(t) \cup M_i(t)| \leq |Q_i(t)| + |M_i(t)| \leq f$ as needed.

We are now ready to complete the proof of Lemma 4:

Proof of Lemma 4. By Claim 1, $|Q_i(t') \cup M_i(t')| \leq f$, and by Lemma 3.11, no base registers on servers in $Q_i(t') \cup M_i(t')$ have ever responded to any low-level writes issued after $t_{i-1}$. Thus, there exists a finite run $r''$, which is identical to $r'$ except all servers in $Q_i(t') \cup M_i(t')$ crash immediately after $r$ and each client $c_1, \ldots, c_{i-1}$ fails before any of its covering writes on registers in $Cov(t_{i-1})$ responds. By $f$-tolerance and obstruction freedom, there exists a fair extension $r'' \sigma$ of $r''$ (i.e., $r'' \sigma \notin \langle r'', Ad_i \rangle$) such that $W_i$ returns in $r'' \sigma$. Since $Q_i \cup M_i$ is stable after $r'$, the set of registers precluded by $Ad_i$ from responding in $r'$ is identical to that in $r''$, and by Assumption 1, no write with a missing response is linearized, $r'$ is indistinguishable from $r''$ to $c_i$. Thus, $r'' \sigma \in \langle r, Ad_i \rangle$, and since $\sigma$ includes the return event of $W_i$, $r'' \sigma$ satisfies the lemma.

We next show that in order to guarantee safety in the face of the environment behaving like $Ad_i$, $W_i$ must trigger a low-level write on at least one non-covered register on each server in a set of $2f + 1$ servers. An illustration of the runs constructed in the proof appears in Figure 2.1.

![Figure 2.1: An illustration of the runs constructed in Lemma 5.](image)

Lemma 5. Consider a run $r \in \langle r_{i-1}, Ad_i, t_r \rangle$ where $t_r > t_{i-1}$, consisting of $r_{i-1}$ followed by a complete high-level write invocation by client $c_i \notin C(t_{i-1})$ that returns at time $t_r$. Then, $|\delta(Tr_i(t_r) \setminus Cov(t_{i-1}))| > 2f$.

Proof. Denote $X \doteq \delta(Tr_i(t_r) \setminus Cov(t_{i-1}))$, and assume by contradiction that $|X| \leq 2f$. Let $S_1 = F_i(t_r)$, $S_2 = Q_i(t_r)$, $S_3 = X \cap (F \setminus F_i(t_r))$ and $S_4 = X \setminus (S_1 \cup S_2 \cup S_3)$. Note that $S_1, S_2, S_3, S_4$ are pairwise disjoint, and $X = S_1 \cup S_2 \cup S_3 \cup S_4$. 

We first show that \(|S_1 + S_4| \leq f\). By Lemma 3.6, \(|S_1| \leq f + 1\). However, if \(|S_1| = f + 1\), then by Lemma 3.4, \(|S_1 - |S_2| = f + 1 - |S_2| \leq 1\), and therefore, \(|S_1 + |S_2| \geq 2f + 1\) violating the assumption. Hence, \(|S_1| \leq f\). By Lemma 3.5, \(|S_2| \leq f\). If \(|S_2| = f\), then by assumption, \(|S_1 \cup S_3 \cup S_4| = |S_1| + |S_3| + |S_4| \leq f\), and therefore, \(|S_1 + S_4| = |S_1| + |S_4| \leq f\). And if \(|S_2| < f\), then by Definitions 2.4 and 4, \(|S_4| = 0\). Hence, \(|S_1 + S_4| = |S_1| + |S_4| \leq f\).

Now let \(r'\) be a fair extension of \(r_{i-1}\) consisting of \(t'_i\) steps in which \(r_{i-1}\) is followed by (1) the crash events of all servers in \(S_1 \cup S_4\), and (2) the respond steps of all the covering writes in \(r_{i-1}\) and (and no other steps). Extend \(r'\) with an invocation of a high-level read operation \(R\) by client \(c_{rd}\) at time \(t'_i\). Since \(|S_1 + S_4| \leq f\), by obstruction freedom and \(f\)-tolerance, there exists time \(t_{rd} > t_{i-1}\) at which \(R\) returns. Since \(r'\) is write-sequential, by WS-Safety, \(R\) must return \(v_{i-1}\).

Next, let \(r''\) be an extension of \(r\) consisting of all steps in \(r\) up to the time \(t_r\), followed by (1) the crash events of all servers in the set \(S_1 \cup S_4\), and (2) the respond steps of all covering writes in \(r_{i-1}\) (and no other steps). Let \(t''_i > t_r\) be the number of steps in \(r''\). By Assumption 1, the values that can be read from the base registers in \(\text{Cov}(t_{i-1})\) at time \(t''_i\) in \(r''\) are identical to those that can be read at time \(t'_i\) in \(r'\). Furthermore, by definitions 2.5 and 4, low-level writes triggered on registers in \(\delta^{-1}(S_2 \cup S_3)\) do not respond before \(t_r\) in \(r\). Thus, by Assumption 1, the values that can be read from the base registers in \(\delta^{-1}(S_2 \cup S_3)\) at time \(t'_i\) in \(r'\) are also the same as those that can be read at time \(t''_i\) in \(r''\). Thus, all registers in non-faulty servers at time \(t'_i\) in \(r'\) will appear to the subsequent reads as having the same content as at the time \(t''_i\) in \(r''\).

We now extend \(r''\) by letting client \(c_{rd}\) invoke high-level read \(R\) at time \(t''_i\). Since \(r'\) is indistinguishable from \(r''\) to \(c_{rd}\), and \(R\) has no concurrent high-level operations, we get that \(R\) returns \(v_{i-1}\) in \(r''\). However, since \(W_i\) is the last complete write preceding \(R\) in \(r''\), by WS-Safety, the \(R\)'s return value must be \(v_i \neq v_{i-1}\). A contradiction.

The following corollary follows immediately from Lemma 5, Definitions 2.4 and 4, and the choice of \(|F| = f + 1\):

**Corollary 2.** Consider a run \(r \in \langle r_{i-1}, Ad_i, t_r \rangle\) where \(t_r > t_{i-1}\), consisting of \(r_{i-1}\) followed by a complete high-level write invocation by client \(c_i \notin C(t_{i-1})\) that returns at time \(t_r\). Then, \(|Q_i(t_r)| = f\).

We are now ready to complete the proof of the induction step of Lemma 2:

**Proof of the induction step (Lemma 2).** By Lemma 4, there exists a run \(r \in \langle r_{i-1}, Ad_i, t_r \rangle\), \(t_r > t_{i-1}\), in which \(r_{i-1}\) is followed by a complete high-level write invocation \(W_i\) by client \(c_i \neq c_{i-1}\) writing a value \(v_i \neq v_{i-1}\) and returning at time \(t_r\). By Corollary 2, \(|Q_i(t_r)| = f\), and therefore, by Lemma 3.4, \(|F_i(t_r)| = f + 1\). Since \(F_i(t_r) \subseteq F\) and \(|F| = f + 1\), we conclude that \(F_i(t_r) = F\). Hence, by Definition 2.6, \(M_i(t_r) = \emptyset\), which by Definition 2.7, implies that \(G_i = \emptyset\). Thus, by Definition 3, no writes on the registers in \(\delta^{-1}(F)\) triggered after \(t_{i-1}\) are blocked.
Hence, by Definition 4.3(a), there exists an extension \( r' \in \langle r, Ad_i, t' \rangle \), for some \( t' \geq t_r \), such that \( \delta(Cov_i(t')) \cap F = \emptyset \). We now show that \( r_i = r' \) and \( t_i = t' \) satisfy the lemma. By the induction hypothesis and the construction of extension \( r', t' \) is a write-only failure-free sequential extension of \( r_{i-1} \) ending at time \( t' \) that consists of \( i \) complete high-level writes of values \( v_1, \ldots, v_i \) by \( i \) distinct clients \( c_1, \ldots, c_i \). It remains to show that the implications (a)--(e) hold for \( t_i = t' \):

a) \( \lvert Cov(t') \rvert \geq i f \): By the induction hypothesis \( \lvert Cov(t_{i-1}) \rvert \geq (i - 1) f \), and by Definition 4, \( Cov(t_{i-1}) \subseteq Cov(t') \). Therefore, we left to show that \( \lvert Cov(t') \setminus Cov(t_{i-1}) \rvert \geq f \). Since by Corollary 2, \( |Q_i(t_r)| = f \), and by Lemma 3.2, \( Q_i(t_r) \subseteq Q_i(t') \), we get \( |Q_i(t')| = f \). By Definition 2.4, \( |Cov_i(t')| \geq |Q_i(t')| \), and by Definition 2.3, \( |Cov(t') \setminus Cov(t_{i-1})| = |Cov_i(t')| \). Therefore, we get \( |Cov(t') \setminus Cov(t_{i-1})| \geq f \).

b) \( \delta(Cov(t')) \cap F = \emptyset \): By the induction hypothesis we get that \( \delta(Cov(t_{i-1})) \cap F = \emptyset \), and by construction of \( r' \) we get that \( \delta(Cov_i(t')) \cap F = \emptyset \). By Definition 2.3, \( Cov(t') = Cov_i(t') \cup Cov(t_{i-1}) \). Therefore, \( \delta(Cov(t')) \cap F = \emptyset \).

\[
\square
\]

Resource Complexity. We will now use Lemma 2 to characterize the minimum resource complexity of the algorithms implementing an \( f \)-tolerant obstruction-free WS-Safe \( k \)-register as a function of the number \( |S| \) of available servers. First, it is easy to see that if \( |S| \leq 2f \), then no such algorithm can exist. This result is implied by an extended statement of Lemma 2 (see Theorem 12 in Appendix ??), and can also be shown directly by a straightforward application of a partitioning argument as discussed in [55, 14]. If \( |S| > 2f \), then we have the following:

**Theorem 2.** For all \( k > 0, f > 0 \), let \( A \) be an \( f \)-tolerant algorithm emulating an obstruction-free WS-Safe \( k \)-register using a collection \( S \) of servers such that \( |S| \geq 2f + 1 \). Then, \( A \) uses at least \( k f + \lceil \frac{kf}{|S|-(f+1)} \rceil (f+1) \) base registers (i.e., \( |\delta^{-1}(S)| \geq k f + \lceil \frac{kf}{|S|-(f+1)} \rceil (f+1) \)).

**Proof.** Let \( G \subseteq S \) be the set consisting of all servers that store at least \( \lceil \frac{kf}{|S|-(f+1)} \rceil \) base registers (i.e., \( \forall s \in G, |\delta^{-1}(\{s\})| \geq \lceil \frac{kf}{|S|-(f+1)} \rceil \) and \( \forall s \in S \setminus G, |\delta^{-1}(\{s\})| < \lceil \frac{kf}{|S|-(f+1)} \rceil \)). We first show that \( |G| \geq f + 1 \). Suppose toward a contradiction that \( |G| < f + 1 \), and pick a set \( F \), such that \( |F| = f + 1 \) and \( S \supset F \supset C \). By Lemma 2.a)-b), there exists a run \( r \) of \( A \) consisting of \( f \) steps such that \( |Cov(t)| \geq k f \) and \( \delta(Cov(t)) \cap F = \emptyset \). Thus, by the pigeonhole argument, and since \( |S \setminus F| = |S| - (f + 1) \), there is at least one server in \( S \setminus F \) that stores at least \( \lceil \frac{kf}{|S|-(f+1)} \rceil \) base registers. A contradiction.

We get that \( |\delta^{-1}(G)| \geq \lceil \frac{kf}{|S|-(f+1)} \rceil (f+1) \). Now, again by Lemma 2.a)-b), there exists a run \( r' \) of \( A \) consisting of \( f' \) steps such that \( |Cov(t')| \geq k f \) and \( \delta(Cov(t')) \cap G = \emptyset \),
meaning that \(|\delta^{-1}(S \setminus G)| \geq kf\). Therefore, we get that \(|\delta^{-1}(S)| \geq kf + \left\lceil \frac{kl}{n - f - 1} \right\rceil (f + 1)\).

The following bound on the number of registers required to emulate a single (i.e., non-fault-tolerant) max-register is a direct consequence of Theorem 2:

**Theorem 3** (Resource Complexity of \(k\)-max-register). For all \(k > 0\), any algorithm implementing a wait-free \(k\)-writer max-register from a collection of wait-free MWMR atomic base registers uses at least \(k\) base registers.

**Proof.** Suppose to the contrary that there exists an algorithm \(A\) implementing a \(k\)-writer max-register using \(\ell < k\) base MWMR wait-free atomic registers. Consider a fault-prone shared memory system consisting of \(n = 2f + 1\) servers each of which stores \(\ell\) MWMR wait-free atomic registers. Run \(n\) copies \(A_1, \ldots, A_n\) of \(A\), one on each server, to obtain \(n = 2f + 1\) copies of \(k\)-writer max-register. Run a generic protocol of [64] to obtain an \(f\)-tolerant emulation \(A\) of a wait-free \(k\)-writer regular register. By assumption, the resource complexity of \(A\) is \((2f + 1)\ell < (2f + 1)k\) base registers. However, by Theorem 2, for \(n = 2f + 1\), it must be at least \(kf + k(f + 1) = (2f + 1)k\). A contradiction.

In Appendix .1, we prove an extended statement of Lemma 2, and use it to show three additional lower bounds as discussed above.

### 2.3.3 Upper Bound

In this section we present an \(f\)-tolerant construction emulating a wait-free WS-Regular \(k\)-register for all combinations of values of the parameters \(k > 0\), \(f > 0\), and \(n\) where \(n > 2f\). Our construction is carefully crafted to deal with the adversarial behaviour (Definition 4) that was exploited in the proof of Lemma 2 while minimizing the resource complexity. Similarly to multi-writer ABD [41, 64, 57], our algorithm uses read and write quorums to read from and write to registers. However, since RMW objects are replaced with read/write registers, and covering low-level writes belonging to old writes can overwrite registers at any time, the quorums in our case must have a larger intersection.

Let \(z = \left\lceil \frac{n-(f+1)}{f} \right\rceil\) and \(y = zf + f + 1\), we construct a collection \(\mathcal{R}\) of \(m = \left\lceil \frac{k}{z} \right\rceil\) disjoint sets \(R_0, \ldots, R_{m-1}\), each of which consist of \(y\) registers, and if \(k\) is not an integer, then we add to \(\mathcal{R}\) another disjoint set \(R_m\) of \((k - \left\lfloor \frac{k}{z} \right\rfloor) f + f + 1\) registers. Intuitively, \(z\) is the maximum number of writers that can be supported by a single set of \(y\) registers as can be deduced from Lemma 2’s argument. If \(z\) divides \(k\), then exactly \(k/z\) such sets are needed to accommodate the total of \(k\) writers. Otherwise, the remaining \(k \mod z\) writers are moved to an overflow set \(R_m\). Note that for all \(R_i \in \mathcal{R}\), \(2f + 1 \leq |R_i| \leq n\). Then, we distribute the registers in each set \(R_i\) on servers in \(S\) so that every register in \(R_i\) is mapped to a different server (i.e., \(|\delta(R_i)| = |R_i|\)). Figure 2.2 demonstrates a possible mapping from registers to servers. All in all, we use \(\sum_{R_i \in \mathcal{R}} |R_i| = \left\lceil \frac{k}{z} \right\rceil y + (k - \left\lfloor \frac{k}{z} \right\rfloor) f + (f + 1)\left(\left\lfloor \frac{k}{z} \right\rfloor - \left\lfloor \frac{k}{z} \right\rfloor\right) = \cdots = kf + \left\lfloor \frac{k}{z} \right\rfloor (f + 1) = kf + \left\lfloor \frac{k}{f + 1} \right\rfloor (f + 1)\) registers.
2.3. Resource Complexity of Write-Sequential $k$-register Emulation

The resulting layout is then used to derive the read and write quorums as follows: for every set $R_i \in \mathcal{R}$, any subset of $R_i$ of size $|R_i| - f$ is a write quorum for all writers $c_j$ such that $j = \lfloor \frac{i}{z} \rfloor$; and any subset of registers consisting of all registers mapped to $n - f$ servers is a read quorum (i.e., the set of the read quorums is $\{B \subseteq S : \exists S \in \mathcal{S} \text{ s.t. } |S| = 2f + 1 \land \delta^{-1}(S) = B\}$). Observe that by construction of $\mathcal{R}$, for every set $R_i \in \mathcal{R}$, (1) the number of clients mapped to write quorums in $R_i$ is $\left\lfloor \frac{|R_i|-(f+1)}{f} \right\rfloor = \left\lfloor \frac{|R_i|-(f+1)}{f} \right\rfloor$, and (2) any write quorum in $R_i$ intersects with any read quorum on at least $|R_i| - f$ registers. Therefore, in a write-sequential run, the latest written value is always guaranteed to be available to subsequent reads provided every writer $c$ executing a high-level write $W$ leaves no more than $f$ pending low-level writes upon $W$’s completion. To enforce the latter, $c$ is precluded from triggering new low-level writes on registers on which it still has writes pending from preceding high-level writes invocations. In addition, since the registers in every (read or write) quorum are mapped to exactly $n - f$ servers, each quorum access is guaranteed to terminate, and thus, the algorithm is wait-free. The algorithm’s pseudo-code appear in Algorithm 2.

Figure 2.2: A possible mapping from $\mathcal{R}$ to $\mathcal{S}$ in case $n = 6, k = 5$, and $f = 2$. 
Algorithm 2 \( \forall f > 0 \forall k > 0, \forall n = |S| \geq 2f + 1. \)

**Types:**

\( TSVa\)l = \( \mathbb{N} \times \mathbb{V} \), with selectors \( ts \) and \( val \).

\( States = TSVa\)l \( \times 2^{TSVal} \times 2^S \times 2^R \) with selectors \( tsVal, rdSet, wrSet \) and \( coverSet \).

**Base Objects and Servers:**

\( \forall b \in \delta^{-1}(S), b \in TSVa\)l, initially, \( (0, v_0) \).

Let \( z \triangleq \lceil n/(f+1) \rceil, y \triangleq zf + f + 1 \), and \( m \triangleq \lfloor \frac{z}{2} \rfloor \).

\( R = \{ R_0, \ldots, R_{m-1} \} \subset \delta^{-1}(S) \) s.t.

1. \( \forall i \in \{0, \ldots, m-2\}, |R_i| = y \). If \( \lceil \frac{y}{z} \rceil = \lceil \frac{y}{z} \rfloor \), then \( |R_{m-1}| = y \). Else, \( |R_{m-1}| = (k - \lceil \frac{y}{z} \rceil)f + f + 1 \).

2. \( \forall R_i, R_j \in R, R_i \cap R_j = \emptyset \).

3. \( \forall R_i \in R, |\delta(R_i)| = |R_i| \).

**Clients states:**

\( \forall i \in [k], State_i \in States, \) initially, 

\( \langle (0, 1), v_0, \emptyset, R_i, \emptyset \rangle \), where \( j = \lceil \frac{y}{z} \rceil \).

**Code for client \( c_i \), \( 1 \leq i \leq k \):**

1. **operation** \( WRITE(v) \)
2. \( value \leftarrow collect() \)
3. \( State_i.tsVal.val \leftarrow v \)
4. \( State_i.tsVal.ts \leftarrow value.ts + 1 \)
5. \( j \leftarrow \lceil \frac{z}{2} \rceil \)

---

**Algorithm 2**

\( \forall f > 0 \forall k > 0, \forall n = |S| \geq 2f + 1. \)

**Types:**

\( TSVa\)l = \( \mathbb{N} \times \mathbb{V} \), with selectors \( ts \) and \( val \).

\( States = TSVa\)l \( \times 2^{TSVal} \times 2^S \times 2^R \) with selectors \( tsVal, rdSet, wrSet \) and \( coverSet \).

**Base Objects and Servers:**

\( \forall b \in \delta^{-1}(S), b \in TSVa\)l, initially, \( (0, v_0) \).

Let \( z \triangleq \lceil n/(f+1) \rceil, y \triangleq zf + f + 1 \), and \( m \triangleq \lfloor \frac{z}{2} \rfloor \).

\( R = \{ R_0, \ldots, R_{m-1} \} \subset \delta^{-1}(S) \) s.t.

1. \( \forall i \in \{0, \ldots, m-2\}, |R_i| = y \). If \( \lceil \frac{y}{z} \rceil = \lceil \frac{y}{z} \rfloor \), then \( |R_{m-1}| = y \). Else, \( |R_{m-1}| = (k - \lceil \frac{y}{z} \rceil)f + f + 1 \).

2. \( \forall R_i, R_j \in R, R_i \cap R_j = \emptyset \).

3. \( \forall R_i \in R, |\delta(R_i)| = |R_i| \).

**Clients states:**

\( \forall i \in [k], State_i \in States, \) initially, 

\( \langle (0, 1), v_0, \emptyset, R_i, \emptyset \rangle \), where \( j = \lceil \frac{y}{z} \rceil \).

**Code for client \( c_i \), \( 1 \leq i \leq k \):**

1. **operation** \( WRITE(v) \)
2. \( value \leftarrow collect() \)
3. \( State_i.tsVal.val \leftarrow v \)
4. \( State_i.tsVal.ts \leftarrow value.ts + 1 \)
5. \( j \leftarrow \lceil \frac{z}{2} \rceil \)

---

The registers store values in \( \mathbb{V} \) each of which is associated with a unique timestamp.

(Note that since safety is required only in write-sequential runs, we do not need to break ties with clients’ ids.) To write a value \( v \) to the emulated register, a client \( c_i \) first accesses a read quorum (via \( collect() \) in lines 22–26) and selects a new timestamp \( ts \) which is higher than any other timestamp that has been returned. It then proceeds to trigger low-level writes of \( \langle ts, v \rangle \) on registers in \( R_j = \lceil \frac{y}{z} \rceil \), so as to ensure that (1) \( \langle ts, v \rangle \) is stored in a write quorum \( wq \) (lines 8–11), and (2) no more than \( f \) registers in \( R_j \) are left covered by \( c_i \)'s
writes (the current and the previous operations). The latter is achieved by preventing \( c_i \) from triggering a new low-level write on any register that has not yet responded to the previously triggered one (lines 9–10). To read a value, a client simply reads all registers in a read quorum, via \( \text{collect}() \), and returns the value having the highest timestamp.

**Correctness proof**

The space complexity of the algorithm is \( \Sigma_{R_i \in R} |R_i| = \left\lceil \frac{k}{2} \right\rceil y + (k - \left\lceil \frac{k}{2} \right\rceil) f + (f + 1)(\left\lceil \frac{k}{2} \right\rceil - \left\lfloor \frac{k}{2} \right\rfloor) = \cdots = kf + \left\lceil \frac{k}{2} \right\rceil (f + 1) = kf + \left\lceil \frac{k}{2} \right\rceil \left( \frac{k - 1}{2} \right) \right\rfloor \right\rfloor \) registers. Below, we prove that the algorithm satisfies wait-freedom and write-sequential regularity. The following observation follows from code and the construction of the sets in \( R \); (1) writers never trigger low-level writes on base object with pending low-lever writes from previous \( \text{writes} \), (2) writers wait for \( n - f \) base objects to reply (line 24), and for every set \( R_i \in R \), the number of client that write to registers in \( R_i \) is \( \left\lceil \frac{|R_i| - (f + 1)}{f} \right\rceil = \frac{|R_i| - (f + 1)}{f} \).

**Observation 3.** For every \( 0 < i \leq k \) for every time \( t \) in a run \( r \), if writer \( c_i \) have no pending \( \text{WRITE} \) at \( t \) then it covers at most \( f \) base objects at time \( t \).

**Lemma 6.** Consider a write-sequential run \( r \) of the algorithm, and consider two sequential \( \text{writes} \) \( W_i, W_j \) in \( r \) s.t. \( W_i \) precedes \( W_j \). Then \( W_j \)'s value is associated with a bigger timestamp than \( W_i \)'s value.

**Proof.** Since \( W_i \) precedes \( W_j \), \( W_i \) starts the \( \text{collect} \) in line 2 after \( W_i \) returns. \( W_j \) triggers low level writes with its value and timestamp on base objects in \( R_i \) (\( l = \left\lfloor \frac{k}{2} \right\rfloor \)) that are not covered by its previous \( \text{writes} \), and waits for \( |R_i| - f \) low level writes to respond (line 11) before it returns. Thus, since \( |\delta(R_i)| = |R_i| \), \( W_j \) starts its \( \text{collect} \) after \( W_i \) writes its timestamp to at least \( |R_i| - f \) base objects in different servers, none of which is covered by low-level write of \( W_i \) previous \( \text{writes} \).

Moreover, since the number of writers excluding \( W_i \) that write to base objects in \( R_j \) is \( \left\lceil \frac{|R_i| - (f + 1)}{f} \right\rceil - 1 = \frac{|R_i| - (f + 1)}{f} - 1 \), readers do not write, and each writer covers at most \( f \) base objects, we get that at least \( f + 1 \) servers has a base object that stores \( W_i \)'s timestamp when \( W_j \) begins its \( \text{collect} \). Now since \( \text{collect} \) reads all base object in at least \( n - f \) servers (line 24), \( W_j \) sees \( W_i \)'s timestamp and picks a bigger one (line 4).

**Corollary 3.** Consider a write-sequential run \( r \) of the algorithm. If \( \text{WRITE} \) \( W_i \) precedes \( \text{WRITE} \) \( W_j \), then \( W_j \) is associated with a bigger timestamp than \( W_i \).

**Lemma 7.** Consider a write-sequential run \( r \) of the algorithm, and a read operation \( rd \) and a \( \text{WRITE} \) \( W \) in \( r \). Let \( ts \) be the timestamp associated with \( W \). If \( W \) precedes \( rd \), then \( rd \) returns a value associated with timestamp \( ts' \geq ts \).

**Proof.** Let \( t \) be the time when \( W \) returns, and assume w.l.o.g that \( W \) is performed by client \( c_i \) s.t. \( \left\lfloor \frac{k}{2} \right\rfloor = j \). Before \( W \) returns it \( c_i \) triggers low level writes with its value and
timestamp on base objects in $R_j$ that are not covered by its previous WRITEs, and waits for $|R_j| - f$ low level writes to respond. The number of clients excluding $c_i$ that trigger low-level writes on base objects is $R_j$ is $\left\lfloor \frac{|R_j| - (f + 1)}{f} \right\rfloor - 1 = \frac{|R_j| - (f + 1)}{f} - 1$, and by Observation 3, each client covers at most $f$ base objects at time $t$. By Corollary 3 and since readers do not write, every low level write in $r$ that is triggered after time $t$ is associated with a bigger timestamp than $ts$. Therefore, since $|\delta(R_j)| = |R_j|$, there is a set of $f + 1$ base objects, each of which mapped to a different server, s.t. at any time $t' \geq t$ the timestamp each of them stores is bigger than or equal to $ts$.

Since $W$ precedes $rd$, $rd$ starts the collect after time $t$. And since collect reads all base object in at least $n - f$ servers, $rd$ sees at least one value associated with timestamp bigger than or equal to $ts$, and thus, returns a value associated with timestamp $ts' \geq ts$.

**Definition 5.** For every write-sequential run $r$, for every read $rd$ in $r$ that returns a value associated with timestamp $ts$ we define the sequential run $\sigma_{rd}$ as follows: All the completed write operations in $r$ are ordered in $\sigma_{rd}$ by their timestamp, and $rd$ is added after the WRITE operation that is associated with $ts$.

In order to show that the algorithm simulates a write-sequential regular register we need to prove that for every write-sequential run $r$, for every read $rd$, $\sigma_{rd}$ preserves the real time order of $r$ and the sequential specification. Note that the sequential specification is satisfied by construction, and we prove the real time order in the next lemma.

**Lemma 8.** For every write-sequential run $r$, for every complete read $rd$ that returns a value associated with timestamp $ts$ in $r$, $\sigma_{rd}$ preserves $r$’s operation precedence relation (real time order).

**Proof.** By Corollary 3, the real time order of $r$ between every two WRITE operations is preserved in $\sigma_{rd}$. We left to show that the real time order of $r$ between $rd$ and any WRITE $W$ in $\sigma_{rd}$ is preserved. Consider two cases:

- $W$ precedes $rd$ in $r$. By Lemma 7, $W$ is associated with a timestamp smaller than or equal to $ts$, and thus, by construction of $\sigma_{rd}$ the real time order between $rd$ and $W$ is preserved.

- $rd$ precedes $W$ in $r$. Let $W_{ts}$ be the WRITE operation associated with timestamp $ts$. Since $rd$ returns a value associated with timestamp $ts$, $W_{ts}$ starts before $rd$ completes, and since $r$ is write-sequential, $w_{ts}$ precedes $W$ in $r$. Thus, by lemma 6, $W$ is associated with bigger timestamp than $ts$. Therefore, by construction of $\sigma_{rd}$ the real time order between $rd$ and $W$ is preserved.

**Corollary 4.** For every write-sequential run $r$, for every complete read $rd$ in $r$, there is a linearization of $rd$ and all the WRITE operations in $r$. 

\[ \square \]
2.4 Discussion and Future Directions

We introduced a new hierarchy, which classifies object types by the number of base objects of a given type required to emulate an $f$-tolerant register, as a function of the number of writers $k$ and the number of available servers $n$. Interestingly, our hierarchy can be used to derive resource complexity bounds in the standard shared memory model (i.e., without object failures) as evidenced by our proof of a lower bound on the number of registers required for implementing a max-register for $k$ writers. Our main technical contribution comprises the lower bound of $\left\lceil \frac{k}{n-(f+1)\frac{f}{f+1}} \right\rceil (f+1) + kf$ and the upper bound of $\left\lfloor \frac{k}{n-(f+1)\frac{f}{f+1}} \right\rfloor (f+1) + kf$ on the resource complexity of emulating an $f$-tolerant $k$-writer register from $n$ fault-prone servers storing read/write registers. To strengthen our lower bound, it was proved for emulations satisfying weak liveness and safety properties.

Future directions. First, for some choices of $k$ and $n$, our bounds leave a small gap that can be closed. Second, an interesting question that arises is whether our lower bound is tight for stronger properties. In the special case of $n = 2f + 1$ servers, emulation with stronger regularity [65] is possible with $(2f + 1)k$ registers (tight to our lower bound). However, the question of the general case ($n \geq 2f + 1$) remains open. In addition, since atomicity usually requires readers to write, it is interesting to investigate whether the space complexity (assuming read/write registers) in this case also linearly depends on the number of readers.

Another possible direction is to extend the hierarchy with more types (e.g., multiple assignment), and to also consider the time complexity of the emulations. For example, we showed that although a max-register can be implemented from a single CAS, the time complexity of the implementation is high. An interesting open question is to determine whether this tradeoff is inherent.
Chapter 3

Space Bounds for Reliable Storage: Fundamental Limits of Coding

In recent years we have seen an exponential increase in storage capacity demands, creating a need for big data storage solutions. In this era, distributed storage plays a key role. Data is typically stored on a collection of nodes accessed asynchronously by clients over a network. By storing redundant information, data remains available following failures. The most common approach to achieve this is via replication [10]; in asynchronous settings, $2f + 1$ replicas are needed in order to tolerate $f$ failures [10]. Given the immense size of data, the storage cost of replication is significant. Previous works have attempted to mitigate this cost via the use of erasure codes [6, 21, 42, 23, 77, 30].

Indeed, codes can reduce the storage cost as long as data is not accessed concurrently by multiple clients. For example, if the data size is $D$ bits and a single failure needs to be tolerated, erasure-coded storage ideally requires $(k + 2)D/k$ bits for some parameter $k > 1$ instead of the $3D$ bits needed for replication. But as concurrency grows, the cost of erasure-coded storage grows with it: when $c$ clients access the storage concurrently, existing asynchronous code-based algorithms [21, 42, 23, 30] store $O(cD)$ bits in storage nodes or communication channels. Intuitively, this occurs because coded data cannot be reconstructed from a single storage node. Therefore, writing coded data requires coordination – old values cannot be deleted before ensuring that sufficiently many blocks of the new value are in place. This is in contrast to replication, where written values can always be read coherently from a single copy, and so old values may be safely overwritten without coordination.

In this Chapter we prove that this extra cost is inherent: Given three problem parameters: $f$, $c$, and $D$, where $f$ is the number of storage node failures tolerated (client failures are unrestricted), $c$ is the concurrency allowed by the algorithm, and $D$ is the data size,
we prove that the storage complexity is $\Theta(\min(f, c) \cdot D)$. Asymptotically, this means either a storage cost as high as that of replication, or as high as keeping as many versions of the data as the concurrency level.

**Lower bound** Our results are proven for emulations of a lock-free multi-reader multi-writer regular register [52, 65]; see Section 3.1 for definitions. Interestingly, the lower bound does not hold for the weaker safe register semantics; in Section 3.5 we present a simple storage-efficient wait-free algorithm that ensures safe semantics, but not regularity. We consider algorithms that use (arbitrary) black-box encoding schemes, i.e., produce and manipulate code blocks of a given value independently of other values and meta-data; as formalized in Section 3.1.2. The storage consists of such code blocks, in addition to possibly unbounded data-independent meta-data, (e.g., timestamps), which we do not count as part of the storage cost. Our black-box assumption excludes storage-reduction techniques like de-duplication, which do require data-dependent meta-data. In Section 3.2 we survey how this assumption holds in related work on popular storage algorithms [6, 21, 42, 23, 30, 43], and compare it with assumptions made in proving other lower bounds [22]. Yet, the question whether there is a more storage-efficient algorithm that circumvents our result by taking stored values into consideration remains open; see further discussion in Section 3.6.

We prove the bound in Section 3.3: we first use a fundamental pigeonhole argument to show that as long as no ongoing write operation contributes code blocks consisting of $D$ or more bits to the storage, no write operation can complete. We then define a parameter $0 < \ell \leq D$. For a given $\ell$, we devise a particular adversary behavior, which we prove drives the storage to a state where either (1) $f + 1$ storage nodes hold at least $\ell$ bits each, or (2) the storage holds more than $D - \ell + 1$ bits in distinct code blocks for each of $c$ different operations. Now, picking $\ell = D/2$ implies our lower bound.

**Algorithm** To prove our bound tight, we present in Section 3.4 a reliable storage algorithm whose storage cost is $O(\min(f, c) \cdot D)$. We achieve this by combining the advantages of replication and erasure coding. Our algorithm does not assume any a priori bound on concurrency; rather, it uses erasure codes when concurrency is low and adaptively switches to replication when it is high.
3.1 Model

3.1.1 Preliminaries

We consider an asynchronous fault-prone shared memory system [4, 1, 48] consisting of set \( B = \{bo_1, \ldots, bo_n\} \) of \( n \) base objects (typically residing at distinct storage nodes) supporting arbitrary atomic read-modify-write (RMW) access by clients from some infinite set \( \Pi \) (see Figure 3.1). Any \( f \) out of \( n \) base objects and any number of clients may fail by crashing, for some predefined \( f < n/2 \).

![Figure 3.1: Clients and base objects.](image)

An algorithm defines the behavior of clients as deterministic state machines, where state transitions are associated with actions such as RMW trigger/response. A configuration is a mapping to states from system components, i.e., clients and base objects. An initial configuration is one where all components are in their initial states.

A run of algorithm \( A \) is a (finite or infinite) alternating sequence of configurations and actions, beginning with some initial configuration, such that configuration transitions occur according to \( A \). For a run \( r \), \( \text{trace}(r) \) is the subsequence of \( r \) consisting of all the operation invocation and returns in \( r \). We use the notion of time \( t \) during a run \( r \) to refer to the configuration reached after the \( t \)th action in \( r \). A run fragment is a contiguous subsequence of a run starting and ending with a configuration. We assume that runs are well-formed, in that each client’s first action is an invocation, and a client has at most one outstanding operation at any time.

We say that a base object or client is faulty in a run \( r \) if it fails any time in \( r \), and otherwise, it is correct. A run is fair if (1) for every RMW triggered by a correct client on a correct base object, there is eventually a matching response, (2) every correct client gets infinitely many opportunities to trigger RMWs.

We study algorithms that emulate a shared register [52], which stores a value \( v \) from some domain \( V \), where \( D = \log_2 |V| \). Initially, the register holds some initial value \( v_0 \in V \). Clients interact with the emulated register via high-level read and write operations. A client that performs a write operation is called a writer, and a client performing a read is a reader.
To distinguish the high-level emulated operations from low-level base object access, we refer to the latter as RMWs. We say that RMWs are triggered and respond, whereas operations are invoked and return. A (high-level) operation is emulated via a series of trigger and respond actions on base objects, starting with the operation’s invocation and ending with its return. In the course of an operation, a client triggers RMWs separately on each $bo_i \in B$. The state of each $bo_i \in B$ changes atomically, according to the RMW triggered on it, at some point after the time when the RMW is triggered but no later than the time when the matching response occurs. To distinguish incomplete invocations to the emulated register from incomplete RMWs triggered on base objects, we refer to the former as outstanding operations and to the latter as pending RMWs.

A parameter $c$ defines the write concurrency level, that is, at most $c$ write operations are outstanding at a given time. We next define the safety and liveness properties we use in this chapter.

**Liveness** There is a range of possible liveness conditions, which need to be satisfied in fair runs. A *wait-free* object is one that guarantees that every correct client’s operation completes, regardless of the actions of other clients. A *lock-free* object guarantees progress: if at some point in a run there is an outstanding operation of a correct client, then *some* operation eventually completes. An *FW-terminating* [1] register is one that has *wait-free* write operations, and in addition, if there are finitely many *write* invocations in a run, then every *read* operation completes.

**Safety** In order to define regularity, we first introduce some terminology: Operation $op_i$ precedes operation $op_j$ in a run $r$, denoted $op_i \prec_r op_j$, if $op_j$’s return occurs before $op_i$’s invoke in $r$. Operations $op_i$ and $op_j$ are concurrent in a run $r$ if neither one precedes the other. A run with no concurrent operations is *sequential*. Two runs are *equivalent* if every client performs the same sequence of operations in both, where operations that are outstanding in one can either be included in or excluded from the other. A linearization of a run $r$ is an equivalent sequential run that preserves $r$’s operation precedence relation and the object’s sequential specification. The sequential specification for a register is as follows: A read returns the latest written value, or $v_0$ if none was written. A write $w$ in a run $r$ is relevant to a read $rd$ in $r$ [65] if $rd \not\prec_r w$; *rel-writes*($r, rd$) is the set of all *writes* in $r$ that are relevant to $rd$.

Following Lamport [52], we consider a hierarchy of safety notions. Lamport [52] defines regular and safe single-writer registers. Shao et al. [65] extend Lamport’s notion of regularity to MWMR registers, and give four possible definitions. Here we use two of them. The first is the weakest definition, and we use it in our lower bound proof. The second, which we use for our algorithm, is the strongest definition that is satisfied by ABD [10] in case readers do not change the storage (no *write-back*):

A MWMR register is *weakly regular* (called *MWRegWeak* in [65]), if for every run $r$ and read $rd$ that returns in $r$, there exists a linearization of the subsequence of $r$ consisting of
The coding scheme is based on two functions: the encoding function $E$ maps value/natural number pairs to code blocks. We denote the number of bits in block $e$ as $|e|$. The decoding function $D: 2^E \rightarrow V \cup \{\bot\}$ takes as a parameter a set of code blocks and returns a value in $V$, or $\bot$ in case no value can be decoded. For example, in a replication approach, each block $e$ can be a full value $v$, so $D(\{e\})$ simply returns $v$. Another example is $k$-of-$n$ erasure codes, where for any value $v$ and any subset $S$ of size $k$ of the set $\{e_i \mid e_i = E(v,i), 1 \leq i \leq n\}$, $D(S) = v$. We capture rateless codes [62], in which an encoder can generate a limit-less sequence of blocks, by using $\mathbb{N}$ as the domain for block numbers.

We encapsulate the encoder and decoder into two oracles, $oracle_E$ and $oracle_D$ as illustrated in Figure 3.2. The interaction with these oracles is as follows:

**Definition 6 (Encoding/Decoding Oracles).** A $w$=write($v$) (read()) invocation at a client $c_k$ initializes an oracle$_E(c_k, w)$ (oracle$_D(c_k, w)$, resp.), which expires when $w$ completes. oracle$_E(c_k, w)$ exposes a get($i$) operation, which returns $E(v,i)$ for $i \in \mathbb{N}$; and oracle$_D(c_k, w)$ exposes two operations, push($e$, $i$) and done($i$), such that for all $i \in \mathbb{N}$, if $c_i$ calls done($i$), then its read operation completes and returns $D(\{e \mid push(e,i) \text{ previously occurred}\})$. We omit the parameters $c_k,w$ when they are clear from the context.

Writers produce code blocks via oracle$_E$ and store them in the storage, whereas readers try to obtain enough blocks to decode legal values via oracle$_D$. In addition to code blocks, clients and base objects can store unbounded meta-data, e.g., program counters and timestamps. But to avoid trivializing the problem, the meta-data must be data-independent, as formally defined below.

Information is represented as list of code blocks and meta-data, $\langle e_1, e_2, \ldots, e_l; m \rangle$, where $\forall i, e_i \in E$ and the meta-data $m$ is from some arbitrary domain. The state of a
3.1. Model

(a) A writer and its oracle.

(b) Run \( r_o \)

Figure 3.2: A model for code-based storage. Encoding and decoding are captured by oracles.

client that has an outstanding operation consists of the information stored at the client as well the parameters of its pending RMWs that have not yet taken effect. The state of a client with no outstanding operation is empty. A base object’s state consists of the information stored at the base object and all the responses of pending RMWs that took effect on it. For a base object \( bo \) (client \( ci \)), we denote the list of code blocks in \( bo \)’s (\( ci \)’s) state at time \( t \) in run \( r \) as \( bo_r(t) \) (resp. \( ci_r(t) \)).

Let \( S \) be an ordered set including all base objects and clients, i.e., \( B \cup \Pi \) ordered in some arbitrary way. For \( S = \{ bo_1, . . . , bo_k, ci_1, . . . \} \subseteq S \), \( S'(t) \) is the list of lists \( bo_1(t), \ldots , bo_k(t), ci_1(t), \ldots \) sorted according to their order in \( S \). A block instance \( b \in S'(t) \) is a triple \( ⟨ i, j, e ⟩ \) so that \( e \) is stored in the \( j \)th position in the \( i \)th list in \( S \). We refer to the block contents as \( b.e \).

Storage cost We count the number of bits stored in blocks in base objects as well as in clients, and neglect meta-data size. Note that oracle states are not counted as part of the storage cost, since we wish to measure the additional space required for making the data available for shared access, beyond its (trivial) existence at its sources and readers.

Definition 7 (Storage Cost). The storage cost at time \( t \) in a run \( r \) is \( \Sigma_{b \in S'(t)} |b.e| \). The storage cost of an algorithm \( A \) is the maximum storage cost at any point \( t \) in any run \( r \) of \( A \).

Assumptions To make sure that the encoding does not leak information using block sizes, we assume symmetry, in the sense that output block sizes do not depend on input values. (Otherwise, we could for example, represent three values 0, 1, and 10 using a single coded block \( e_1 \) of size at most 1 bit by having \( |e_1| = 0 \) encode 10). Formally:

Definition 8 (Symmetric Encoding). An encoding function \( E \) is symmetric if for every \( v, v' \in V \) and for all \( i \in \mathbb{N} \), \( |E(v, i)| = |E(v', i)| \). We denote size\((i) \overset{Δ}{=} |E(v, i)| \). Note that different block numbers (of all values) may have different sizes.
We next state our assumption that the storage treats the coding as a black-box. First, we define the notion of a source function, which we shall use to prohibit generation of code blocks by any source other than \(\text{oracle}_E\):

**Definition 9 (Source Function).** A function is a source function for a run \(r\) if it maps every \((b, t)\) s.t. \(b \in S'(t)\) to a pair \((w, i)\) s.t. \(b.e\) was returned by \(\text{get}(i)\) in \(\text{oracle}_E(w)\).

We use a source function to trace blocks in the storage to operations that produced them. To capture the restriction that the algorithm’s decision what to store does not rely on block contents, we stipulate that we can replace the value written by a write operation \(w\) in a run \(r\) by an arbitrary value \(v\), yielding the same sequence of states and actions, except that all stored block instances whose source is \((w, i)\) are replaced with \(\text{oracle}_E(v, i)\). For clarity, we refer to the operation as \(w\) in both runs (see Figure 3.3).

**Definition 10 (Black-Box Coding).** An algorithm \(A\) is black-box coding if for every run \(r\) there is a source function \(\text{source}'\) s.t. for every \(w = \text{write}(u)\) operation in \(r\), \(\forall u \in \mathcal{V}\), there is run \(r_v\) satisfying the following:

1. \(r_v\) has the same sequences of invocations and returns as \(r\) except that \(w = \text{write}(v)\) (possibly with no change) and return values of read operations may be different; and
2. client and base object states at every time \(t\) in \(r_v\) are the same as at time \(t\) in \(r\) except that the contents of every \(b \in S'(t)\) s.t. \(\text{source}'(b, t) = (w, i)\) for some \(i\) is replaced by \(e' = \text{oracle}_E(v, i)\).

In the following, we will only consider source functions satisfying Definition 10. In case multiple such source functions for \(r\) exist, we fix an arbitrary one and refer to it as \(\text{source}'\).

### 3.2 Related work

Our model captures numerous existing distributed storage algorithms, including ones that use replication \([10]\), and erasure codes \([6, 21, 42, 23, 30, 43]\). We note that some...
of them report a storage cost below $O(cD)$ [6, 21, 77, 30]. This is sometimes achieved by assuming periods of synchrony [6]. Other works shift the cost from storage nodes to the network and keep unbounded information in channels [30, 21]. However, since we define parameters and responses of pending RMWs to be part of clients’ and base objects’ states, information in channels is counted in our storage cost model and hence these algorithms are subject to our bound. The only non-black-box storage algorithm we are aware of is [77], where multiple values are encoded jointly, saving space, but also forfeiting regular register semantics. It is as of now unclear whether lifting the black-box assumption suffices in order to circumvent our result.

In [68] we showed a special case of the result in this chapter for infinite concurrency. Cadambe et al. [22] prove closely related lower bounds for coded storage algorithms. First, they show that asynchronous fault-tolerant storage algorithms require strictly more storage than synchronous erasure-coded algorithms. Second, similarly to this work, they extend the result given in In [68] to show that the storage cost must grow linearly with $\min(f, c)$, but their result is proven under a different set of assumptions than ours. In particular, while both papers make certain “black box” assumptions about the storage, [22] does not rule out joint coding as we do, but instead restricts protocol actions in a way that forbids them from depending on a written value in more than one communication round; this affords the protocol more freedom than our model in one communication round, and less freedom in all other rounds. On the face of it, the two sets of assumptions appear to be incomparable, though they both achieve the same end result, as we discuss in Section 3.6 below. Additionally, our bound allows algorithms to use unbounded (data-independent) meta-data and is proven for lock-free register emulations, whereas the bound in [22] includes meta-data and is shown for wait-free registers.

In Chapter 2 we showed that the number of fault-prone read/write registers needed to emulate a reliable multi-writer register grows linearly with the number of clients that can write to the register (even in sequential runs). Here, on the other hand, we consider storage nodes supporting fully general read-modify-write, for which that lower bound does not apply.

The challenge of providing a lower bound on stored data when meta-data is potentially unbounded was also previously addressed in the context of byzantine storage [25]. That paper has shown that certain storage algorithms cannot be “amnesic”, i.e., cannot “forget” values written to them. Like our black-box assumption, the notion of amnesia was defined in terms of runs. However, it did not yield explicit bound on storage cost.

### 3.3 Storage Lower Bound

We now show a lower bound of $O(\min(f, c) \cdot D)$ bits on the storage cost of any lock-free algorithm that uses symmetric black-box coding to simulate a weakly regular register:

**Theorem 5.** Consider a lock-free algorithm $A$ that uses symmetric black-box coding to simulate a weakly regular register. The storage cost of $A$ is $\Omega(\min(f, c) \cdot D)$. 
For the sake of our proof, we quantify the number of bits in blocks contributed by client \( c_i \)'s operation \( w \) to base objects and clients other than \( c_i \).

**Definition 11.** Let \( S \subset \mathcal{S} \), and consider a time \( t \) and an operation \( w \) by client \( c_j \) in a run \( r \). We define \( S'(t, w) \equiv \{ i \in \mathbb{N} \mid \exists b \in (S \setminus \{ c_j \})'|(t): \text{source}'(b, t) = \langle w, i \rangle \} \), and \( |S'(t, w)| \equiv \Sigma_{i \in S'(t, w)} \text{size}(i) \).

For \( I \subseteq \mathbb{N} \), we say that two values \( v' \neq v'' \) in \( \mathbb{V} \) are \( \ell \)-colliding if \( \forall i \in I \), \( \mathbb{E}(v', i) = \mathbb{E}(v'', i) \). We next use the pigeonhole argument and the assumption of symmetric black-box coding in order to show that write operations cannot return until some write stores enough bits in different blocks in every set of \( n \) \( f \) base objects.

**Claim 2.** Let \( w \) be a write operation invoked in a run \( r \) of \( A \), and \( t \) be a point in \( r \). Consider a set of values \( U \subset \mathbb{V} \), \( |U| < 2^{D-1} \), and a set of base objects \( S \subset \mathcal{S} \). If \( |S'(t, w)| < D \), then there are two \( S'(t, w) \)-colliding values \( u \neq u' \) in \( \mathbb{V} \setminus U \).

**Proof.** Since \( |\mathbb{V} \setminus U| > 2^{D-1} \) and \( |S'(t, w)| < D \), the claim follows from the pigeonhole argument.

**Lemma 9.** Consider a run \( r \) of algorithm \( A \) that begins with the invocation of \( c \) concurrent write operations. Let \( S \) be a set of at least \( n \) \( f \) base objects and assume that at every time \( t \) in \( r \) for every operation \( w \) in \( r \), \( |S'(t, w)| < D \). Then no write operation returns in \( r \).

**Proof.** Let \( \mathcal{W}_{ops} = \{ w_1, \ldots, w_c \} \) be the set of \( c \) concurrent writes invoked in \( r \). Assume by contradiction that there exists a complete write in \( \mathcal{W}_{ops} \). Let \( w \) be the first such write, and \( t \) be the time when it returns. Next we inductively build a sequence of sets of values \( U_0, U_1, \ldots, U_c \), where \( |U_i| = i \):

- \( U_0 = \{ \} \)
- \( \forall i \in \{0, \ldots, c-1\} \), use \( U_i \) to build \( U_{i+1} \). By the lemma premise, \( |S'(t, w_{i+1})| < D \). Now since \( |U_i| < c < 2^{D-1} \), by Claim 2, there are two \( S'(t, w_{i+1}) \)-colliding values \( u_{w_{i+1}} \neq u'_{w_{i+1}} \) in \( \mathbb{V} \setminus U_i \). We let \( U_{i+1} = U_i \cup \{ u_{w_{i+1}} \} \).

The set \( U_c \) contains exactly \( c \) (different) values s.t. for every operation \( w_i \in \mathcal{W}_{ops} \) there is a value \( u_{w_i} \in U_c \) that has a \( S'(t, w_i) \)-colliding value \( u'_{w_i} \) in \( \mathbb{V} \). By applying Definition 10 (\( c \) times), there is a run \( r' \) that begins with the invocation of \( c \) concurrent write operations, in which every operation \( w_i \in \mathcal{W}_{ops} \) writes \( u_{w_i} \) s.t. \( w \) returns at time \( t \), and for every operation \( w_i \in \mathcal{W}_{ops} \), \( S'(t, w_i) = S'(t, w_i) \). Next, let clients with outstanding operations and all base objects in \( B \setminus S \) fail at time \( t \) in \( r' \) (note that by assumption \( |S| \geq n-f \), so \( |B \setminus S| \leq f \)), and let some client \( c_j \) invoke a solo read operation at time \( t+1 \). By lock-freedom, \( c_j \)'s read operation completes, and by regularity, it returns a value \( u \in U_c \) at some time \( t' > t \).

Let \( w' \) be the operation that writes \( u \) in \( r' \). Since \( u \) has a \( S'(t, w') \)-colliding value \( u' \) and since \( S'(t, w') = S'(t, w') \), \( u \) and \( u' \) are \( S'(t, w') \)-colliding. By Definition 10, there is
a run \( r'' \) with the same operations as in \( r' \) except that \( w' \) writes \( u' \) (instead of \( u \)) s.t.
\( \) every client’s and base object’s state at time \( t \) in \( r' \) is identical to its state at time \( t \) in \( r'' \) (note that clients with outstanding operations and all base objects in \( B \setminus S \) fail at time \( t \) except that for every block instance \( b \in S' (t) \) s.t. \( \text{source}^r (b, t) = \langle w', i \rangle \), \( b.e \) is replaced with a block \( E(u', i) \). In particular, states of base objects in \( S \) at time \( t \) in \( r' \) are identical to their states at time \( t \) in \( r'' \) (note that clients with outstanding operations and all base objects in \( B \setminus S \) fail at time \( t \) except that for every block instance \( b \in S' (t) \) s.t. \( \text{source}^r (b, t) = \langle w', i \rangle \), \( b.e \) is replaced with a block \( E(u', i) \).

Now since \( u \) and \( u' \) are \( S_r (t, w') \)-colliding, states of base objects in \( S \) at time \( t \) in \( r'' \) are identical to their states at time \( t \) in \( r' \). In addition, since clients with outstanding operations and all base objects in \( B \setminus S \) fail at time \( t \), the solo reader \( c_j \) cannot distinguish between \( r' \) and \( r'' \), and thus, it pushes the same blocks to its oracle and calls \texttt{done} with the same number in \( r'' \) as in \( r' \), and therefore, its read operation returns \( u \) at time \( t'' \) in run \( r'' \). However, since the clients invoke write operations with different values in \( r' \), \( u \) is not written in \( r'' \). A contradiction to weak regularity.

Having shown a condition under which write operations cannot complete, we define an (unfair) adversary behavior that takes advantage of this in order to prevent progress. We introduce some notation, and then use it in order to define the adversary. We define a parameter \( 0 < \ell \leq D \), and for any time \( t \) in a run \( r \) of algorithm \( A \) we define the following sets, as illustrated in Figure 5.2. For convenience, from now on we omit the superscript \( r \).

- \( C(t) \): the set of outstanding write operations at time \( t \).
- \( C^{-}_\ell (t) = \{ w \in C(t) \mid ||S(t, w)|| \leq D - \ell \} \): The set of write operations each of which has at most \( D - \ell \) bits in blocks, produced by its oracle with different numbers, in the storage (excluding the client performing it) at time \( t \).
- \( C^+_\ell (t) = C(t) \setminus C^-_\ell (t) \).
- \( F_\ell (t) = \{ b_{o_i} \in B \mid \sum_{b \in \{ b_{o_i} \}(t)} |b.e| \geq \ell \} \). Base objects that store blocks that consist (together) of more than \( \ell \) bits at time \( t \). These are base objects that we will “freeze” in our counter-example because they are already “full”, i.e., consume enough space for our lower bound.

We fix the parameter \( \ell \) throughout the proof and omit subscript \( \ell \) from the notation. The next observation on storage cost immediately follows from the definitions.

**Observation 4.** At any point \( t \) in every run \( r \) of \( A \), the storage cost is at least \( |C^+(t)|(D - \ell + 1) \).

We next define a particular adversary behavior that schedules actions in a way that prevents progress. Note that the adversary controls the scheduling of client actions and RMW responses.
Figure 3.4: Example scenario in run of a storage algorithm with adversary $Ad$. In this example, $2D/5 < \ell < D$. At time $t$, only $w_2$ and $w_4$ are in $C^-(t)$, where $w_2$ has no pending RMWs and $w_2$ has one triggered RMW on $b_1 \in F(t)$ and one triggered RMW on $b_3 \notin F(t)$. Therefore, by the first rule, $Ad$ schedules the response on the RMW triggered by $w_2$ on $b_3$. In this example $w_2$ overwrites $w_3$’s block in $b_3$, thus $w_3$ moves from $C^-$ to $C^+$. Then, at time $t+1$, no response can be scheduled by rule 1 (no operation in $C^-(t+1)$ has a pending RMW on a base object in $N \setminus F(t+1)$), so by rule 2, $Ad$ chooses $w_2$ and lets it trigger an RMW on base object $b_2$. Now since $w_2$ is the only operation that has a pending RMW on a base object not in $F(t+2)$, $Ad$ schedules the response on the RMW triggered by $w_2$ on $b_2$ at time $t+2$. In this example $w_2$ adds a block with $\ell$ bits to $b_2$. Thus, $c_2$ is included in $C^+(t+3)$. In addition, $b_2$ stores more than $\ell$ bits at time $t+3$, so it belongs to $F(t+3)$.

**Definition 12.** ($Ad$) At any time $t$, $Ad$ schedules an action as follows:

1. If there is a pending RMW on a base object in $B \setminus F(t)$ by a client performing an operation in $C^-(t)$, then choose the longest pending of these RMWs, allow it to take effect on the corresponding base object, and schedule its response.

2. Else, choose in a fair order an operation by a client $c_i \in \Pi$ and schedule its action (trigger RMW, call its oracle, get response from its oracle, or return), without allowing it to affect the base object yet. By fair order we mean any order in which every client is chosen infinitely often (e.g., $c_1, c_2, c_3, c_1, c_2, c_3 \ldots$).

In other words, $Ad$ delays RMWs triggered by operations in $C^+(t)$ (for which the storage already holds $D - \ell$ bits) as well as RMWs on “frozen” base objects in $F(t)$ (which store at least $\ell$ bits), and fairly schedules all other actions. We demonstrate $Ad$’s behavior in Figure 5.2. Though this behavior may be unfair, in every infinite run of $Ad$, every correct client gets infinitely many opportunities to take steps. We use $Ad$ to build an unfair run with no progress (no write returns), and then build an indistinguishable fair run to contradict lock-freedom. The following observation immediately follows from the adversary’s freezing of base objects in $F$.

**Observation 5.** Assume run $r$ of algorithm $A$ in which the environment behaves like $Ad$. For each base object $bo$, if $bo \in F(t)$ at some time $t$, then $bo \in F(t')$ for all $t' > t$ in $r$. 
3.3. Storage Lower Bound

Another consequence of \(Ad\)'s behavior is captured by the following:

**Lemma 10.** Consider a run \(r\) of algorithm \(A\). If the adversary behaves like \(Ad\), then for every time \(t\) and for every write operation \(w\) in \(r\), \(||(S \setminus F(t))(t,w)|| < D\).

**Proof.** Assume by way of contradiction that there is time \(t\) and write operation \(w\) performed by client \(c_j\) s.t. \(||(S \setminus F(t))(t,w)|| ≥ D\). The definition of \((S \setminus F(t))(t,w)\) takes into account only blocks returned by \(w\)'s oracle that are stored outside of \(c_j(t)\). Thus, \(w\) triggered at least one RMW that has a matching response before time \(t\) in \(r\). Let \(t' ≤ t\) be the time when the last RMW triggered by \(w\) responded, and denote this RMW by \(rmw\) and the base object on which \(rmw\) was triggered by \(bo\). By \(Ad\), \(w, w ∈ C^-(t' − 1)\), and therefore, by definition, \(||(S \setminus F(t' − 1))(t' − 1,w)|| ≤ D − ℓ\). Now consider two cases:

- First, \(rmw\) adds blocks (possibly overwriting other blocks) with less than \(ℓ\) bits to \(bo\). In this case, since \(bo\) is the only storage component that changed at time \(t'\), \(||(S \setminus F(t'))(t',w)|| < D\).
- Second, \(rmw\) adds blocks (possibly overwriting other blocks) with at least \(ℓ\) bits to \(bo\). In this case, \(bo ∈ F(t')\). Now since \(||(S \setminus F(t' − 1))(t' − 1,w)|| ≤ D\), by Observation 5, \(F(t' − 1) ⊆ F(t')\), and given \(bo ∈ F(t')\) and it is the only storage component that changed at time \(t'\), we get \(||(S \setminus F(t'))(t',w)|| ≤ ||(S \setminus F(t' − 1))(t' − 1,w)|| < D\).

So far we showed that \(||(S \setminus F(t'))(t',w)|| < D\). By Observation 5, and since no RMW by \(w\) takes effect after time \(t'\), \((S \setminus F(t''))(t'',w) ⊆ (S \setminus F(t'))(t',w), ∀t'' ≥ t'\). Therefore, we get \(||(S \setminus F(t))(t,w)|| < D\). A contradiction.

\(\Box\)

The next corollary uses Lemmas 9 and 10 in order to conclude that \(Ad\) can prevent progress of write operations.

**Corollary 5.** Consider a run \(r\) of algorithm \(A\) that begins with the invocation of \(c\) concurrent write operations. If the adversary behaves like \(Ad\) and \(|F(t)| ≤ f\) for all \(t\) in \(r\), then no write operation returns in \(r\).

**Proof.** By Lemma 10, for every time \(t\) for every write operation \(w\) in \(r\), \(||(S \setminus F(t))(t,w)|| < D\). And since \(B ⊆ S\), for every time \(t\) for every write operation \(w\) in \(r\), \(||(B \setminus F(t))(t,w)|| < D\). Now since \(|F(t)| ≤ f\) for every time \(t\) in \(r\), \(|B \setminus F(t)| ≥ n − f\). Therefore, by Lemma 9, no write operation returns in \(r\).

\(\Box\)

We have shown that \(Ad\) can prevent completion of write operations in algorithms that store \(ℓ\) bits in less than \(f + 1\) base objects. However, this does not directly imply a storage bound, since \(Ad\) is not fair. In the next lemma we close this gap by showing a fair run in which lock-freedom must be satisfied, i.e., operations invoked by correct clients must eventually complete, in order to blow up the storage. We show that for every algorithm,
we can build a run where at some point the algorithm either stores $\ell$ bits in each of $f + 1$ base objects (namely, $\exists t : |F(t)| > f$), or there are $c$ concurrent operations each of which adds at least $D - \ell + 1$ bits to the storage cost (i.e., $|C^+(t)| = c$).

**Lemma 11.** There is a run $r$ of $A$ and a time $t$ in $r$ when $|C^+(t)| = c$ or $|F(t)| > f$.

**Proof.** Assume by way of contradiction that there is no such run of algorithm $A$. We first build a run $r$ of $A$ with $c$ clients that concurrently write different values, in which the environment behaves like adversary $Ad$. By the contradiction assumption, $|C^+(t)| < c$ and $|F(t)| \leq f$ for all $t$ in $r$. We start with the invocation of $c$ concurrent write operations, and allow the run to proceed indefinitely according to $Ad$. We say that a client $c$, which performs write operation $w$, is stuck in $r$ if there is a time $t$ in $r$ s.t. for all $t' \geq t$, $w \in C^+(t')$ (and so no RMWs triggered by $c$ take effect after time $t$). By Observation 5 and the assumption that $|F(t)| \leq f$ for all $t$, there is a time $t_1$ in $r$ s.t. for every time $t_2 \geq t_1$, $F(t_1) = F(t_2)$.

Now we build a run $r'$ that is identical to $r$ but every base object $bo \in F(t_1)$ fails at time $t_1 (|F(t_1)| \leq f)$, and every stuck client fails after its last RMW takes effect. Since by $Ad$, RMWs do not take effect on base objects in $F(t_1)$ after time $t_1$, runs $r$ and $r'$ are indistinguishable to all correct clients and base objects. Now notice that by $Ad$’s behavior, each correct client in $r'$ gets infinitely many opportunities to trigger RMWs. In addition, since (1) for every correct client $c_i$ in $r'$ there are infinitely many times $t$ when $c_i \in C^-(t)$, (2) $Ad$ picks responses from base objects not in $F(t)$ in the order they are triggered, and (3) there are no correct base objects in $F(t')$ for all $t' > t_1$, every RMW triggered by a correct client on a correct base object has a matching response in $r'$. Therefore, run $r'$ is fair.

By the contradiction assumption $|C^+(t)| < c$ for all $t$ in $r$. Therefore, there is at least one client that is not stuck in $r$, and thus, there is at least one client that is correct in $r'$. Hence, by lock-freedom, some client eventually completes its write operation in $r'$. Now since $r$ and $r'$ are indistinguishable to all clients that are correct in both, the same is true in $r$. However, by Corollary 5, no write operation completes in $r$. A contradiction.

So far we have shown that every algorithm has a run where at some point either $\ell$ bits are stored in $f + 1$ base objects, or there are $c$ concurrent operations each of which adds at least $D - \ell + 1$ bits to the storage cost. We now combine this result with Observation 4 to conclude our lower bound:

**Proof (Theorem 5).** Let $\ell = D/2$. By Lemma 11, there is a run $r$ of $A$ and a time $t$ in $r$ when $|C^+(t)| = c$ or $|F(t)| > f$. If $|F(t)| > f$, then the storage cost at time $t$ in $r$ is $(f + 1)\ell = (f + 1)D/2 = \Omega(fD)$. Otherwise, $|C^+(t)| = c$, and so by Observation 4, the storage cost at time $t$ in $r$ is at least $c(D - \ell) = cD/2 = \Omega(cD)$. The theorem follows.
By picking $\ell = D$, we get a second conclusion from Lemma 11 and Observation 4. The following corollary proves that any coding scheme short of full replication must exhibit storage growth linear in the concurrency.

**Corollary 6.** The storage cost of any algorithm that uses a black-box coding scheme to simulate a weakly regular lock-free register, and does not store $D$ bits (enough to represent a full replica) in $f + 1$ base objects, grows linearly with the concurrency.

### 3.4 Adaptive Regular Register

We present here a storage algorithm that combines full replication with erasure coding in order to achieve the advantages of both.

#### 3.4.1 Algorithm

**Erasure codes.** A $k$-of-$n$ erasure code takes a value from $\mathbb{V}$ and produces a set $S$ of $n$ blocks from $\mathcal{E}$ s.t. the value can be restored from any subset of $S$ that contains no less than $k$ different blocks. We assume that the size of each block is $D/k$. $\text{Oracle}_E$ and $\text{Oracle}_D$ are encapsulated by two functions $\text{encode}$ and $\text{decode}$, respectively: $\text{encode}$ gets a value $v \in \mathbb{V}$ and returns a set of $n$ ordered elements $W = \{\langle e_1, 1 \rangle, \ldots, \langle e_n, n \rangle\}$, where $e_1, \ldots, e_n \in \mathcal{E}$, and $\text{decode}$ gets a set $W' \subseteq \mathcal{E} \times \mathbb{N}$ and returns $v' \in \mathbb{V}$ s.t. if $|W'| \geq k$ and $W' \subseteq W$, then $v = v'$. We use $k = n - 2f$. Note that when $k = 1$, we get full replication.

The main idea behind our algorithm is to have base objects store blocks from at most $k$ different writes, and then turn to store full replicas. Our algorithm satisfies strong regularity and FW-termination. In the next section we prove the following:

**Theorem 6.** There is an FW-terminating algorithm that simulates a stringly regular register, whose storage cost is $\min((c + 1)(2f + k)D/k, (2f + k)2D)$ bits. Moreover, in a run with a finite number of writes, if all the writers are correct, the storage is eventually reduced to $(2f + k)D/k$ bits.

Notice that $k$ is a parameter of the algorithm, and if we pick $k = f$, then asymptotically the storage cost of our algorithm is $O(\min(cD, fD)) = O(\min(c, f) \cdot D)$.

The algorithm’s pseudocode appears in Algorithms 3-5. The algorithm uses a set of $n$ shared base objects $bo_1, \ldots, bo_n$ each of which holds three fields $V_p$, $V_f$, and $\text{storedTS}$. 
Algorithm 3 Definitions.

1: `TimeStamps = N × Π`, with selectors `num` and `c`, ordered lexicographically.

2: `Pieces = (E × N)`

3: `Chunks = Pieces × TimeStamps`, with selectors `val`, `ts`

4: `encode : V → 2E×{1,2,...,n} → V s.t. ∀v ∈ V, encode(v) = \{(∗, 1), ..., (∗, n)\} ∧ ∀W ∈ 2E×N, if W ⊆ encode(v) ∧ |W| ≥ k, then decode(W) = v`

5: `base objects:`

6: `∀ i ∈ {1, ..., n}, bo_i = ⟨storedTS, V_p, V_f⟩ s.t. V_f, V_p ⊂ Chunks, and storedTS ∈ TimeStamps, initially ⟨⟨0, 0⟩, {⟨0, 0⟩, {v_0, i}⟩}, {}⟩.`

The `V_p` field holds a set of timestamped code blocks so that the `i^{th}` block of a value can be stored in the `V_p` field of object `bo_i`. The `V_f` field stores a timestamped replica of a single value, (represented as a set of `k` code blocks). And `storedTS` holds a timestamp, as explained below.

**Write operation and storage efficiency**    

The write operation (lines 3–14) consists of 3 sequentially executed rounds: `read timestamp`, `update`, and `garbage collection`; and, the read consists of one or more sequentially executed `read` rounds. At each round, the client invokes RMWs on all base objects in parallel, and awaits responses from at least `n – f` base objects. The read rounds of both write and read rely on the `readValue` routine (lines 21–28) to collect the contents of the `V_p` and `V_f` fields from `n – f` base objects, as well as to determine the highest `storedTS` known to these objects. The implementations of the update and garbage collection rounds are given by the update (lines 29–36) and GC (lines 37–42) routines, respectively.

The write implementation starts by encoding `v` into `k` code blocks (line 4) and invoking the read round where the client uses the combined contents of the `V_p`, `V_f` and `storedTS` fields returned by `readValue` to determine the timestamp `ts` to be stored alongside `v`’s code blocks on the base object; `ts` is set to be higher than all returned timestamps thus ensuring that the order of the timestamps associated with the stored values is compatible with the order of their corresponding writes, (which is essential for regularity).

The client then proceeds to the update round where it attempts to store the `i^{th}` code block `⟨e, i⟩` of `v` in `bo_i.V_p` if the size of `bo_i.V_p` is less than `k` (lines 33), or its full replica in `bo_i.V_f` if `ts` is higher than the timestamp associated with the value currently stored in `bo_i.V_f` (line 35). Storing `⟨e, i⟩` in `bo_i.V_p` coincides with an attempt to reduce its size by removing stale code blocks of values whose timestamps are smaller than `storedTS` (line 33). This guarantees that the size of `V_p` never exceeds the number of concurrent writes, which is a key for achieving our adaptive storage bound. Lastly, the client updates `bo_i.storedTS` so as its new value is at least as high as the one returned by the `readValue` routine. This allows the timestamp associated with the latest complete update to propagate to the base object being written, in order to prevent future writes of old blocks into this base object.

In the write’s garbage collection round, the client attempts to further reduce the stor-
age usage by (1) removing all code blocks associated with timestamps lower than $ts$ from both $b_i.V_p$ and $b_i.V_f$ (lines 38–39), and (2) replacing a full replica (if it exists) of its written value $v$ in $b_i.V_f$ with its $i$th code block $\langle e, i \rangle$ (line 41). It is safe to remove the full replica and values with older timestamps at this point, since once the update round has completed, it is ensured that the written value or a newer written value is restoreable from any $n - f$ base objects. This mechanism ensures that all code blocks except the ones comprising the value written with the highest timestamp are eventually removed from all objects’ $V_p$ and $V_f$ sets, which reduces the storage to a minimum in runs with finitely many writes, which all complete. The garbage collection round also updates the $b_i.storedTS$ field to ensure its value is at least as high as $ts$.

### Algorithm 4 regular register emulation.

#### Algorithm for client $c_j$.

1: **local variables:**
2: $storedTS, ts \in TimeStamp, WriteSet \in Pieces$

3: **operation** $Write(v)$
4: $WriteSet \leftarrow encode(v)$
5: **round 1:** read timestamps
6: $\langle storedTS, ReadSet \rangle \leftarrow readValue()$
7: $tmp \leftarrow max(storedTS.num,$
8: $\max\{tmp' \mid \langle \langle tmp', +, + \rangle, + \rangle \in ReadSet\}$
9: $ts \leftarrow (n + 1, j)$
10: **round 2:** update
11: **for** $i = 1$ to $n$
12: $update(b_i, WriteSet, ts, storedTS, i)$
13: **wait** for $n - f$ responses
14: **round 3:** garbage collect
15: **for** $i = 1$ to $n$
16: $GC(b_i, WriteSet, ts, i)$
17: **wait** for $n - f$ responses
18: **return** "ok"

19: **operation** $Read()$
20: $\langle storedTS, ReadSet \rangle \leftarrow readValue()$
21: **while** $\exists ts \geq storedTS$ s.t.
22: $\{|\{ts, v \mid \langle ts, v \rangle \in ReadSet\}| \geq k \}$
23: $\langle storedTS, ReadSet \rangle \leftarrow readValue()$
24: $ts' \leftarrow \max_{ts \geq storedTS}\{|\{ts, v \mid \langle ts, v \rangle \in ReadSet\}| \geq k \}$
25: **return** $decode\{v \mid \langle ts', v \rangle \in ReadSet\}$

### Key Invariant and read operation

The write implementation described above guarantees the following key invariant: at all times, a value written by either the latest complete write or a newer write is available from every set consisting of at least $n - f$ base objects (either in the form of $k$ code blocks in the objects’ $V_p$ fields, or in full from one of their $V_f$ fields). Therefore, a read will always be able to reconstruct the latest completely written or a newer value provided it can successfully retrieve $k$ matching blocks of this value. However, a read round may sample different base objects at different times (that
is, it does not necessarily obtain an atomic snapshot of the base objects), and the number of blocks stored in \( V_p \) is bounded. Thus, the read may be unable to see \( k \) matching blocks of any single new value, as long as new values continue to be written concurrently with the read.

Nevertheless, for FW-Termination, the reads are only required to return in runs where a finite number of writes are invoked. Our implementation of read (lines 15–20) proceeds by invoking consecutive rounds of RMWs on the base objects via the readValue routine. After each round, the reader examines the collection of returned values and timestamps to determine if any value has \( k \) code blocks and is also associated with a timestamp that is at least as high as \( \text{storedTS} \) (line 17). If any such value is found, the one associated with the highest timestamp is returned (line 20). Otherwise, the reader proceeds to invoke another round of base object accesses. Note that returning values associated with older timestamps may violate regularity, since they may have been written earlier than the write with timestamp \( \text{storedTS} \), which in turn may have completed before the read was invoked.

**Algorithm 5** Functions used in regular register emulation.

21: \textbf{procedure} readValue()
22: \quad \text{ReadSet} \leftarrow \{\}, \text{T} \leftarrow \{\}
23: \quad |\| \text{for} i=1 \text{ to } n
24: \quad \quad \text{tmp} \leftarrow \text{read}(b_0_i)
25: \quad \quad \text{ReadSet} \leftarrow \text{ReadSet} \cup \text{tmp} \cdot V_f \cup \text{tmp} \cdot V_p
26: \quad \quad \text{T} \leftarrow \text{T} \cup \{\text{tmp}\cdot \text{storedTS}\}
27: \quad \text{wait for } n - f \text{ responses}
28: \quad \text{return } \langle \max(\text{T}), \text{ReadSet} \rangle
29: \textbf{update}(b_0, \text{WriteSet}, ts, \text{storedTS}, i) \triangleq
30: \quad \text{if } ts \leq b_0.\text{storedTS}
31: \quad \quad \text{return}
32: \quad \quad \text{if } |b_0.V_p| < k
33: \quad \quad \quad \text{b}\_p \leftarrow b_0.V_p \setminus \{\langle ts', v \rangle \in b_0.V_p | ts' < \text{storedTS}\}
34: \quad \quad \quad \cup \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
35: \quad \quad \quad \text{b} \_f \leftarrow \{\langle ts', (e, i) \rangle | (e, i) \in \text{WriteSet}\}
36: \quad \quad \quad \land j \in \{1, \ldots, k\}
37: \quad \quad \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
38: \quad \quad \quad \land j \in \{1, \ldots, k\}
39: \quad \quad \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
40: \quad \quad \quad \land j \in \{1, \ldots, k\}
41: \quad \quad \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
42: \quad \text{b} \_p \leftarrow \{\langle ts', v \rangle \in b_0.V_p | ts' \geq ts\}
43: \quad \text{b} \_f \leftarrow \{\langle ts', v \rangle \in b_0.V_f | ts' \geq ts\}
44: \quad \text{if } (ts, \ast) \in b_0.V_f
45: \quad \quad \text{v} \_f \text{ holds a full replica of my write}
46: \quad \quad \text{v} \_f \text{ keep only one piece of it}
47: \quad \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
48: \quad \text{b} \_p \leftarrow \{\langle ts', v \rangle \in b_0.V_p | ts' \geq ts\}
49: \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
50: \quad \text{b} \_p \leftarrow \{\langle ts', v \rangle \in b_0.V_p | ts' \geq ts\}
51: \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
52: \quad \text{b} \_p \leftarrow \{\langle ts', v \rangle \in b_0.V_p | ts' \geq ts\}
53: \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
54: \quad \text{b} \_p \leftarrow \{\langle ts', v \rangle \in b_0.V_p | ts' \geq ts\}
55: \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
56: \quad \text{b} \_p \leftarrow \{\langle ts', v \rangle \in b_0.V_p | ts' \geq ts\}
57: \quad \text{b} \_f \leftarrow \{\langle ts, (e, i) \rangle | (e, i) \in \text{WriteSet}\}
3.4.2 Correctness Proofs

Note that we prove here that the algorithm satisfies strong regularity and FW-termination, which are stronger safety and liveness properties than the one used in our lower bound.

We start by proving the storage cost.

**Observation 6.** For every run of the algorithm, for every base object $bo_i$, $bo_i.ts$ monotonically increasing.

**Lemma 12.** Consider a run $r$ of the algorithm, and two writes $w_1, w_2$, where $w_1$ writes with timestamp $ts_1$. If $w_1 \prec_r w_2$, then $w_2$ sets its $\hat{ts}$ to a timestamp that is not smaller than $ts_1$.

*Proof.* By Observation 6, for each base object $bo$, $bo.ts$ is monotonically increasing. Therefore, after $w_1$ finishes the garbage collection phase, there is a set $S$ consisting of $n - f$ base objects s.t. for each $bo_i \in S$, $bo_i.ts \geq ts$. Recall that $n = 2f + k$, thus every two sets of $n - f$ base objects have at least one base object in common. Therefore, $w_2$ gets a response from at least one base object in $S$ in its first phase, and thus sets $\hat{ts} = ts'$ s.t. $ts' \geq ts$.

**Lemma 13.** For any run $r$ of the algorithm, for any base object $bo$ at any time $t$ in $r$, $bo.V_p$ does not store more than one piece of the same write.

*Proof.* The writes perform the second phase at most one time on each base object $bo$, and in each update they store at least one piece in $bo.V_p$. And since they does not store in $bo.V_p$ during the third phase, the lemma follows.

**Lemma 14.** Consider a run $r$ of the algorithm in which the maximum number of concurrent writes is $c < k - 1$. Then the storage at any time in $r$ is not bigger than $(2f + k)(c + 1)D/k$ bits.

*Proof.* Recall that we assume that $n = 2f + k$ and the size of each piece is $D/k$. Thus it suffices to show that there is no time $t$ in $r$ s.t. some base object stores more than $c + 1$ pieces at time $t$.

Assume by way of contradiction that the claim is false. Consider the time $t$ when some $bo \in N$ stores $c + 2$ pieces for the first time. Notice that $|bo.V_p| \leq c + 1 < k$ till time $t$, and therefore, $bo.V_p$ does not contain more then one piece from the same write, and $bo.V_f = \bot$ till time $t'$. Now consider the write $w$ that was invoked last among all the writes that store pieces in $bo.V_p$ at time $t$, denote its piece by $p$. Since $bo$ stores $c + 2$ pieces at time $t'$, by Lemma 14, there must be two writes $w_1$ and $w_2$ whose pieces $p_1, p_2$ are stored at time $t$ in $bo.V_p$, and both returns before $w$ is invoked. Denote their timestamps $ts_1$ and $ts_2$, and assume without loss of generality that $ts_1 > ts_2$. By Lemma 12, $w$ sets its $\hat{ts}$ to $ts'$ s.t. $ts' \geq ts_1 > ts_1$. Now consider two cases. First, if $p$ was added before $p_2$, then $bo.ts > ts_2$ when $p_2$ was added. A contradiction. Otherwise, $p$ was added after $p_2$. Thus, $p_2$ was deleted in line 33 of the update when $p$ was added. A contradiction.
Lemma 15. The storage is never more than \((2f + k)2D\) bits at any time \(t\) in any run \(r\) of the algorithm.

Proof. Each base object stores no more than \(2k\) pieces at any time \(t\) in \(r\). The lemma follows.

\[\square\]

Lemma 16. Consider a run \(r\) of the algorithm with finite number of writes, in which all writes correct. Then the storage is eventually reduced to \((2f + k)D/k\) bits.

Proof. Consider a write \(w\) with the biggest timestamp \(ts\) in \(r\). Since \(w\) is correct, and since \textit{writes} are wait-free, \(w\) returns, and eventually performs \textit{free} on every base object. Consider a base object \(bo\) s.t. \(w\) performs \textit{free} on \(bo\) at time \(t\). Notice that \(w\) deletes all pieces with smaller timestamps than \(ts\) and set \(bo.ts = ts\) at time \(t\). Now recall that \(bo\) ignore all updates with timestamp less than \(bo.ts\), and therefore, \(bo\) store only \(w\)'s piece at any time after time \(t\). The lemma follows.

\[\square\]

From Lemmas 14, 15, and 16 we get:

Corollary 7. The storage of the algorithm is bounded by \((2f + k)2D\) bits, and in runs with at most \(c < k\) concurrent writes the storage is bounded by \((c + 1)D/k\) bits. Moreover, in a run with a finite number of writes, if all the \textit{writes} are correct, the storage is eventually reduced to \((2f + k)D/k\) bits.

We next prove the liveness property.

Lemma 17. Consider a fair run \(r\) of the algorithm. Then every write \(w\) invoked by a correct client \(c_i\) eventually completes.

Proof. Consider a correct client \(c_i\). The \textit{write} \(w\) is divided into three phase s.t. in each phase, \(c_i\) invokes operations on all the base objects, and waits for \(n - f\) responses. The run \(r\) is fair, so every action invoked by \(c_i\) on a correct base object eventually returns, and no more than \(f\) base objects fail in \(r\). Therefore, eventually \(c_i\) receives \(n - f\) responses in each of the phases and returns.

\[\square\]

Observation 7. When a piece from \(bo.V_p\) is deleted, \(bo.ts\) is increased.

Lemma 18. If at time \(t\), \(c_i\) completes the second phase of \textit{write} with timestamp \(ts\), then for every \(t' > t\) for every \(S \subseteq N\) s.t. \(|S| \geq n - f\), exist \textit{write} \(w\) with \(ts' \geq ts\) s.t. at least \(k\) pieces of \(w\) are stored in \(S\).

Proof. Consider time \(t'\). Let \(\bar{ts}\) be the highest timestamp written by a \textit{write} \(w\) that completed the second phase by time \(t\). It is sufficient to show the lemma hold for \(\bar{ts}\).

First note that \(\forall bo, bo.ts \leq \bar{ts}\) before time \(t\), because no \textit{write} with a larger timestamp than \(\bar{ts}\) started the third phase. This means that \(w\)'s \textit{update} left at lest one piece in which
bo it occurred. Now consider a set $S$ of $n - f$ base objects, and since $n = 2f + k$, $w$'s update occurred in set $S'$ that contains at least $k$ base objects in $S$.

If $w$ wrote to $V_p$, it was not overwritten by time $t$, because (1) no other write began free with timestamp bigger than $\tilde{t}_S$, and (2) since there is no base object $bo$ s.t. $bo.ts \geq \tilde{t}_S$, no write delete $w$'s piece in the second phase. Therefore if $w$ wrote to $V_p$ in all base objects in $S'$, the lemma holds.

Otherwise, $w$ wrote $k$ pieces to $V_f$ in base objects in some set $S'' \subseteq S'$. Consider two cases: First, there is base object $bo' \in S''$ s.t. some write overwritten $w$'s pieces in $bo'.V_f$ before time $t$. Since there is no write with timestamp bigger than $\tilde{t}_S$ that started the third phase before time $t$, it is guarantee that $k$ pieces with timestamp $ts' > \tilde{t}_S$ stored in $bo'.V_f$ at time $t$, and the lemma holds. Else, since $w$'s pieces stored in $S' \setminus S''$ does not overwritten before time $t$, the lemma holds (no matter if $w$ performed the third phase or not).

\[\Box\]

**Invariant 1.** For any run $r$ of the algorithm, for any time $t$ in $r$, for any set $S$ of $n - f$ base objects. Let $\tilde{t}_S = \max\{bo.ts \mid bo \in S\}$. Then there is a timestamp $ts' \geq \tilde{t}_S$ s.t. there are at least $k$ different pieces associated with $ts'$ in $S$.

**Proof.** We prove by induction. **Base:** the invariant holds at time 0. **Induction:** Assume that the induction holds before the $t^{th}$ action is scheduled, we show that it holds also at time $t$. Assume that the $t^{th}$ action is RMW on a base object $bo$, and consider any set $S$ of $n - f$ base objects. If $bo \notin S$ then the invariant holds. Else, consider the two possible RMW actions:

- The $t^{th}$ action is **update.** If no pieces are deleted, the invariant holds. If $bo.ts$ is increased, then consider the write with timestamp $ts$ that is the the biggest timestamp among all writes that complete the second phase before time $t$. Notice that $bo.ts \leq ts$ at time $t$, and by Lemma 18, the invariant holds. The third option is that a piece $p$ with timestamp $ts' > bo.ts$ of a write $w$ is deleted and $bo.ts$ is not increased. Note that by Observation 7, such piece can be deleted only from $bo.V_f$, and since $p$ is overwritten by $k$ pieces with bigger timestamp, the invariant holds.

- The $t^{th}$ action is **free.** If $bo.ts$ is not changes, then the invariant holds. Else, Consider the write with the biggest timestamp $ts$ among all writes that complete the second phase before time $t$. Note that $bo.ts$ is set to a timestamp $ts' \leq ts$, so by Lemma 18, the invariant holds.

\[\Box\]

**Lemma 19.** Consider a fair run $r$ of the algorithm. If there is a finite number of write invocations in $r$, then every read operation $rd$ invoked by a client $c_i$ eventually returns.

**Proof.** Assume by way of contradiction that $rd$ does not return in $r$. By Lemma 17, the writes are wait-free, and since the number of write invocations in $r$ is finite, there is a
time $t$ in $r$ s.t. no write performs actions after time $t$. Therefore, any read that invokes $readValue()$ procedure after time $t$ receives a set $S$ of values that is stored in a set of $n - f$ base objects at time $t$. By invariant 1, there is a timestamp $ts$ s.t. there is at least $k$ different pieces in $S$ associated with $ts$, and $ts > bo.ts$ for all $bo \in S$. Now since the every correct read $rd$ invokes $readValue()$ infinitely many times in $r$, $rd$ returns. A contradiction.

The next corollary follows from Lemmas 17, 19.

**Corollary 8.** The algorithm satisfies the WF-termination property.

We now prove that the algorithm satisfies strong regularity.

**Definition 13.** For every run $r$, $\sigma_r$ is a sequential run s.t. the writes in $r$ are ordered in $\sigma_r$ by their timestamp, and every read in $r$ that returns a value associate with timestamp $ts$, is ordered in $\sigma_r$ immediately after the write that is associate with timestamp $ts$.

For simplicity we say the that $v_0$ was written by write $w_0$ that associated to timestamp 0 at time 0.

**Lemma 20.** Consider a run $r$, and a read $rd$ that returns a value $v$. Consider also the timestamp $ts'$ that $rd$ obtains in line 19 (Algorithm 4). Then $v$ is the value written by a write associated with timestamp $ts'$ or $v_0$ if $ts' = 0$.

**Proof.** By the code, if $ts' = 0$, then $rd$ returns $v_0$. Now notice that $rd$ obtains at least $k$ different pieces associated with timestamp $ts'$, thus by decode definition, $rd$ returns $v$.

**Corollary 9.** For every run $r$, $\sigma_r$ satisfies the sequential specification.

**Observation 8.** Consider a write $w$ that obtains $ts$ and $\hat{ts}$ in the first phase, then $ts > \hat{ts}$.

**Lemma 21.** For every run $r$, for every two writes $w_1, w_2$ with timestamp $ts_1, ts_2$. If $w_2$ was invoked after $w_1$ finished the second phase, then $ts_1 < ts_2$.

**Proof.** First notice that for every base object $bo$, if a write $w$ overwrites pieces of a write $w'$ in $bo.V_f$, that $w'$ timestamp is bigger than $w'$'s. And by Observation 8, if $w$ deletes $w'$'s piece from $bo.V_p$, then it stores a piece with bigger timestamp than $w'$'s timestamp. Therefore, the maximal timestamp in each base object is monotonically increasing. Now recall that in the second phase $w_1$ performed update on $n - f$ base object, and notice that after $w_1$ performs update on base object $bo$ the maximal timestamp in $bo$ is at lest as big as $ts_1$. Now since two sets of $n - f$ base object have at least one base object in common, $w_2$ picks $ts > ts_1$.

**Lemma 22.** For every run $r$, for every two writes $w_1, w_2$ in $r$, if $w_1 \prec_r w_2$, then $w_2$ is not ordered before $w_1$ in $\sigma_r$. 

3.5 A (Simple) Safe and Wait-free Algorithm

**Proof.** Follows immediately from Lemma 21.

**Lemma 23.** For every run r, for every read rd and write w₁, if rd ⊑ₗ r w₁, then w₁ is not ordered before rd in σᵣ.

*Proof.* Assume that rd returns value that is associated with timestamp ts belonging to some write w, and w₁ is associated with timestamp ts₁. Since rd returns w’s value, w begins the third phase before rd returns. And since w₁ was invoked after rd returns, w₁ was invoked after w’s second phase. Therefore, by Lemma 21, ts₁ > ts, and thus w₁ is ordered after w in σᵣ. Recall that by the construction of σᵣ, rd is ordered immediately after w in σᵣ, hence, rd is ordered before w₁ in σᵣ.

**Lemma 24.** For every run r, for every read rd and write w₁, if w₁ ⊑ₗ rd, then rd is not ordered before w₁ in σᵣ.

*Proof.* Consider a write w₁ with timestamp ts₁ and a read rd s.t. w₁ ⊑ₗ rd. Assume by way of contradiction that rd is ordered before w₁ in σᵣ. Then rd returns a value with a timestamp ts that is associated with a write w that is ordered before w₁ in σᵣ. By the construction of σᵣ, ts₁ > ts. Now since w₁ completed the third phase before rd invoked, and since by Observation 6, for each bo, bo.ts is monotonically increasing, when rd invoked, for every set S of n − f base objects, the maximal bo.ts of all bo ∈ S is bigger than or equal to ts₁, and thus bigger than ts. Therefore rd set ts, in the first phase, to timestamp bigger than ts, and thus does not return w’s value. A contradiction.

The next corollary follows from Corollary 9, and Lemmas 22, 23, 24.

**Corollary 10.** The algorithm simulates a strongly regular register.

The following theorem stems from Corollaries 7, 8, and 10.

**Theorem 7.** There is a FW-terminating algorithm that simulates a strongly regular register, which storage is bounded by (2f + k)2D bits, and in runs with at most c < k concurrent writes, the storage is bounded by (c + 1)D / k bits. Moreover, in a run with a finite number of writes, if all the writes are correct, the storage is eventually reduced to (2f + k)D / k bits.

3.5 A (Simple) Safe and Wait-free Algorithm

We present here a simple storage-efficient algorithm that ensures safe semantics, but not regularity. Although this algorithm has no practical use, it shows that the impossibility result of Section 3.3 does not apply to a weaker safety property.
3.5.1 Algorithm

This algorithm simulates a wait-free and strongly safe MWMR register using erasure codes (see Section 3.4). It stores exactly \( n \) pieces of the data, one in each base object. The algorithm’s definitions we use here are the same as in section 3.4 (Algorithm 3), and the pseudocode of client \( c_j \) can be found in Algorithm 6.

Since memory is fault-prone, actions are triggered in parallel on all base objects. This parallelism is denoted using \( || \) in the code. Operations then wait for \( n - f \) base objects to respond. Recall that \( n = 2f + k \), so every two sets of \( n - f \) base objects have at least \( k \) pieces in common. Thus, if a write completes after storing pieces on \( n - f \) base objects, a subsequent read accessing any \( n - f \) base objects finds \( k \) pieces of the written value (as needed for restoring the value), provided that they are not over-written by later writes.

A write \((v)\) operation (lines 1–8) first produces \( n \) pieces from \( v \) using \( \text{encode} \), then reads from \( n - f \) base objects to obtain a new timestamp, and finally, tries to store every piece together with the timestamp at a different base object. For every base object \( b_o \), \( c_j \) triggers the update RMW function, which overwrites \( b_o \) only if \( c_j \)'s timestamp is bigger than the timestamp stored in \( b_o \).

A read (lines 12–17) reads the values stored in \( n - f \) base objects, and then tries to restore valid data as follows. If \( c_j \) reads at least \( k \) values with the same timestamp, it uses the \( \text{decode} \) function, and returns the restored value. Otherwise, it returns \( v_0 \). The latter occurs only if there are outstanding writes, that had updated fewer than \( n - f \) base objects before the reader has accessed them. Therefore, these writes are concurrent with \( c_j \)'s read, and by the safety property, any value can be returned in this case. The algorithm’s correctness is formally in the next section.

Algorithm 6 Safe register emulation. Algorithm for client \( c_j \).

```
1: operation write(v)
2:     W ← encode(v)
3:     R ← readValue()
4:     ts ← \( \max(\{ts | (ts, *) \in R\}) + 1, j \)
5:     || for all \((v, i) \in W\) on \( b_o \)
6:     update\((b_o, (v, i), ts)\) \(\triangleright\) trigger RMW
7:     wait for \( n - f \) responses
8:     return “ok”
9: update\((b_o, w, ts)\) \(\triangleq\)
10: if \( ts > b_o.ts \)
11: \( b_o \leftarrow \langle w, ts \rangle \)
12: operation read()
13:     R ← readValue()
14:     if \( \exists ts \text{ s.t. } \{v | (ts, v) \in R\} \geq k \)
15:         ts' ← ts s.t. \( \{|v | (ts', v) \in R\} \geq k \)
16:         return decode\(\{v | (ts', v) \in R\}\)
17:     return \( v_0 \)
18: procedure readValue()
19:     R ← \{ \}
20:     || for \( i=1 \) to \( n \)
21:         R = R \cup read\((b_o_i)\)
22:     wait until \(|R| \geq n - f \)
23:     return R
```

3.5.2 Correctness proof

Lemma 25. The storage of the algorithm is \( nD/k \).
3.5. A (Simple) Safe and Wait-free Algorithm

Proof. The size of each piece is \( D/k \). We have \( n \) base objects, and each base object stores exactly one piece.

\[\]

Lemma 26. The algorithm is wait-free.

Proof. There are no loops in the algorithm, and the only blocking instructions are the waits in lines 7 and 22. In both cases, clients wait for no more than \( n - f \) responses, and since no more than \( f \) base objects can fail, clients eventually continue. Therefore, a client that gets the opportunity to perform infinitely many actions completes its operations.

We now prove that the algorithm satisfies strongly safety. We relay on the following single observation.

Observation 9. The timestamps in the base objects are monotonically increasing.

Definition 14. For every run \( r \), we define the sequential run \( \sigma_{w} \) as follows: All the completed write operations in \( r \) are ordered in \( \sigma_{w} \) by their timestamp.

Lemma 27. For every run \( r \), the sequential run \( \sigma_{w} \) is a linearization of \( r \).

Proof. Since \( \sigma_{w} \) has no read operations, the sequential specification is preserved in \( \sigma_{w} \). Thus, we left to show the real time order. For every two completed writes \( w_i, w_j \) in \( r \), we need to show that if \( w_i \prec_r w_j \), then \( w_i \prec_{\sigma} w_j \).

Denote \( w_i \)'s timestamp by \( ts \). By Observation 9, at any point after \( w_i \)'s return, at least \( n - f \) base objects store timestamps bigger than or equal to \( ts \). When \( w_j \) picks a timestamp, it chooses a timestamp bigger than those it reads from \( n - f \) base objects. Since, \( n > 2f \), \( w_j \) picks a timestamp bigger than \( ts \), and therefore \( w_j \) is ordered after \( w_i \) in \( \sigma_{rd} \).

Definition 15. For every run \( r \), for every read \( rd \) that has no concurrent write operations in \( r \), we define the sequential run \( \sigma_{rd} \) by adding \( rd \) to \( \sigma_{w} \) after all the writes that precede it in \( r \).

In order to show that the algorithm simulates a safe register, we proof in Lemmas 28 and 29 that the real time order and sequential specification respectively, are preserved in \( \sigma_{rd} \).

Lemma 28. For every run \( r \), for every read \( rd \) that has no concurrent write operations in \( r \), \( \sigma_{rd} \) preserves \( r \)'s operation precedence relation (real time order).

Proof. By Lemma 27, the order between the writes in \( \sigma_{rd} \) are preserved, and by construction of \( \sigma_{rd} \) the order between \( rd \) and write operations is also preserved.
Lemma 29. Consider a run $r$ and any read $rd$ that has no concurrent writes in $r$. Then $rd$ returns the value written by the write with the biggest timestamp that precedes $rd$ in $r$, or $v_0$ if there is no such write.

Proof. In case there is no write before $rd$ in $r$, since there are also no writes concurrent with $rd$, $rd$ reads pieces with timestamp $(0,0)$ from all base objects, and thus, returns $v_0$. Otherwise, let $w$ be the write associated with the biggest timestamp $ts$ among all the writes invoked before $rd$ in $r$. Let $t$ be the time when $rd$ is invoked. Recall that $rd$ has no concurrent writes, so all the writes invoked before time $t$ complete before time $t$ and store there pieces in $n-f$ base objects unless the base objects already hold a higher timestamp. By Observation 9 and the fact that $w$ has the highest timestamp by time $t$, we get that at time $t$ there are at least $n-f$ base objects that store a piece of $v$. Since $n=2f+k$, every two sets of $n-f$ base objects have at least $k$ base objects in common. Therefore, $rd$ reads at least $k$ pieces of $v$, and thus, restores and returns $v$.

Corollary 11. There exists an algorithm that simulates a safe wait-free MWMR register with a worst-case storage cost of $nD/k = (2f/k + 1)D$.

3.6 Discussion

We studied the inherent space requirements of reliable storage in asynchronous distributed settings. We proved an asymptotic bound of $\Omega(\min(f,c) \cdot D)$ for any storage algorithm using a symmetric black-box coding scheme, which produces code blocks of values independently of other values. We then presented an algorithm that combines replication and erasure codes, whose storage cost is $O(\min(f,c) \cdot D)$.

Our work leaves open questions for future work. First, it is unclear whether the same lower bound still applies when stored bits are allowed to depend on multiple concurrent write values. The main requirement for extending our proof to general coding is a model that correctly accounts for the information stored in the base objects and clients when the clients code jointly. Our black-box assumption rules out such joint coding. Whereas in principle, [22] allows stored information to depend on multiple input values, their assumption that only one round of the protocol depends on the written value essentially forces clients to “forget” the value they used in such joint coding. For example, if the algorithm stores $v_1 + v_2$ instead of either $v_1$ or $v_2$, it cannot reproduce the original values, rendering such joint coding useless. Second, while asymptotically optimal, the constants in our bound are not tight, and it could be interesting to close this gap. Finally, we believe that our model and adversary definitions can yield additional lower bounds.
Chapter 4

On Liveness of Dynamic Storage

Many works in the last decade have dealt with dynamic reliable distributed storage emulation [66, 8, 40, 41, 24, 17, 53, 19, 50, 16, 15, 37, 49, 74]. The motivation behind such storage is to allow new processes (nodes) to be phased in and old or dysfunctional ones to be taken offline. From a fault-tolerance point of view, once a faulty process is removed, additional failures may be tolerated. For example, consider a system that can tolerate one failure: once a process fails, no additional processes are allowed to fail. However, once the faulty process is replaced by a correct one, the system can again tolerate one failure. Thus, while static systems become permanently unavailable after some constant number of failures, dynamic systems that allow infinitely many reconfigurations can survive forever.

Previous works can be categorized into two main types: Solutions of the first type assume a churn-based model [51, 59] in which processes are free to announce when they join the storage emulation [17, 15, 16, 11] via an auxiliary broadcast sub-system that allows a process to send a message to all the processes in the system, (which may be unknown to the sending processes). The second type solutions extend the register’s API with a reconfiguration operation for changing the current configuration of participating processes [66, 8, 50, 37, 40, 41, 24], which can be only invoked by members of the current configuration. In this paper we consider the latter. Such an API allows administrators (running privileged processes), to remove old or faulty processes and add new ones without shutting down the service; once a process is removed from the current configuration, a system administrator may shut it down. Note that in the churn-based model, in contrast, if processes have to perform an explicit operation in order to leave the system (as in [17, 11]), a faulty process can never be removed. In addition, since in API-based models only processes that are already within the system invoke operations, it is possible to keep track of the processes in the system, and thus auxiliary broadcast is not required.

Though the literature is abundant with dynamic storage algorithms in both models, to the best of our knowledge, all previous solutions in asynchronous and eventually
synchronous models restrict reconfigurations in some way in order to ensure completion of all operations. Churn-based solutions assume a bounded churn rate [17, 15, 11], meaning that there is a finite number of joining and removing processes in a given time interval. Some of the API-based solutions [66, 8, 50, 37] provide liveness only when the number of reconfigurations is finite, whereas others discuss liveness only in synchronous runs [40, 41, 24]. Such restrictions may be problematic in emerging highly-dynamic large-scale settings.

Baldoni et al. [15] showed that it is impossible to emulate a dynamic register that ensures completion of all operations without restricting the churn rate in asynchronous churn-based models in which processes can freely abandon the computation without an explicit leave operation. Since a leave and a failure are indistinguishable in such models, the impossibility can be proven using a partition argument as in [10].

In this chapter we revisit this question in the API-based model. First, we prove a similar result for asynchronous API-based dynamic models, in which one unremoved process can fail and successfully removed ones can go offline. Specifically, we show that even the weakest type of storage, namely a safe register [52], cannot be implemented so as to guarantee liveness for all operations (i.e., wait-freedom) in asynchronous runs with an unrestricted reconfiguration rate. Note that this bound does not follow from the one in [15] since a process in our model can leave the system only after an operation that removes it successfully completes.

Second, to circumvent our impossibility result, we define a dynamic failure detector that can be easily implemented in eventually synchronous systems, and use it to implement dynamic storage. We present an algorithm, based on state machine replication, that emulates a strong shared object, namely a wait-free atomic dynamic multi-writer, multi-reader (MWMR) register, and ensures liveness for all operations without restricting the reconfiguration rate. Though a number of previous algorithms have been designed for eventually synchronous models [40, 41, 24, 53, 19], to the best of our knowledge, our algorithm is the first to ensure liveness of all operations without restricting the reconfigurations rate.

In particular, previous algorithms [40, 41, 24, 53, 19] that used failure detectors, only did so for reaching consensus on the new configuration. For example, reconfigurable Paxos variants [53, 19], which implement atomic storage via dynamic state machine replication, assume a failure detector that provides a leader in every configuration. However, a configuration may be changed, allowing the previous leader to be removed (and then fail) before another process $p$ (with a pending operation) is able to communicate with it in the old configuration. Though a new leader is elected by the failure detector in the ensuing configuration, this scenario may repeat itself indefinitely, so that $p$’s pending operation never completes.

We, in contrast, use the failure detector also to implement a helping mechanism, which ensures that eventually some process will help a slow one before completing its own reconfiguration operation even if the reconfiguration rate is unbounded. Such
mechanism is attainable in API-based models since only members of the current configuration invoke operations, and thus helping process can know which processes may need help. Note that in churn-based models in which processes announce their own join, implementing such a helping mechanism is impossible, since a helping process cannot possibly know which processes need help joining.

The remainder of this chapter is organized as follows: In Section 4.1 we present the model and define the dynamic storage object we seek to implement. Our impossibility proof appears in Section 4.2, and our algorithm in Section 4.3. Finally, we conclude the paper in Section 4.4.
4.1 Model

In Section 4.1.1, we present the preliminaries of our model, and in Section 4.1.2, we define the dynamic storage service.

4.1.1 Preliminaries

We consider an asynchronous message passing system consisting of an infinite set of processes $\Pi$. Processes may fail by crashing subject to restrictions given below. Process failure is modeled via an explicit fail action. Each pair of processes is connected by a communication link. A service exposes a set of operations. For example, a dynamic storage service exposes read, write, and reconfig operations. Operations are invoked and subsequently respond.

An algorithm $A$ defines the behaviors of processes as deterministic state machines, where state transitions are associated with actions, such as send/receive messages, operation invoke/response, and process failures. A global state is a mapping to states from system components, i.e., processes and links. An initial global state is one where all processes are in initial states and all links are empty. A send action is enabled in state $s$ if $A$ has a transition from $s$ in which the send occurs.

A run of algorithm $A$ is a (finite or infinite) alternating sequence of global states and actions, beginning with some initial global state, such that state transitions occur according to $A$. We use the notion of time $t$ during a run $r$ to refer to the $t$th action in $r$ and the global state that ensues it. A run fragment is a contiguous subsequence of a run. An operation invoked before time $t$ in run $r$ is complete at time $t$ if its response event occurs before time $t$ in $r$; otherwise it is pending at time $t$. We assume that runs are well-formed [?], in that each process’s first action is an invocation of some operation, and a process does not invoke an operation before receiving a response to its last invoked one.

We say that operation $op_i$ precedes operation $op_j$ in a run $r$, if $op_i$’s response occurs before $op_j$’s invocation in $r$. Operations $op_i$ and $op_j$ are concurrent in run $r$, if $op_i$ does not precede $op_j$ and $op_j$ does not precede $op_i$ in $r$. A sequential run is one with no concurrent operations. Two runs are equivalent if every process performs the same sequence of operations (with the same return values) in both, where operations that are pending in one can either be included in or excluded from the other.

4.1.2 Dynamic storage

The distributed storage service we consider is a dynamic multi-writer, multi reader (MWMR) register [8, 50, 37, 72, 56, 41], which stores a value $v$ from a domain $V$, and offers an interface for invoking read, write, and reconfig operations. Initially, the register holds some initial value $v_0 \in V$. A read operation takes no parameters and returns a value from $V$, and a write operation takes a value from $V$ and returns “ok”. We define $\text{Changes}$ to be the set $\{\text{remove}, \text{add}\} \times \Pi$, and call any subset of $\text{Changes}$ a set of changes. For example,
4.1. Model

Figure 4.1: Notation illustration. \(\text{add}(p)\) (\(\text{remove}(p)\)) represents \(\text{reconfig}(\langle \text{add}, p \rangle)\) (respectively, \(\text{reconfig}(\langle \text{remove}, p \rangle)\)).

\(\{\langle \text{add}, p_3 \rangle, \langle \text{remove}, p_2 \rangle\}\) is a set of changes. A \(\text{reconfig}\) operation takes as a parameter a set of changes and returns “ok”. For simplicity, we assume that a process that has been removed is not added again.

**Notation** For every subset \(w\) of \(\text{Changes}\), the removal set of \(w\), denoted \(w.\text{remove}\), is \(\{p|\langle \text{remove}, p \rangle \in w\}\); the join set of \(w\), denoted \(w.\text{join}\), is \(\{p|\langle \text{add}, p \rangle \in w\}\); and the membership of \(w\), denoted \(w.\text{membership}\), is \(w.\text{join} \setminus w.\text{remove}\). For example, for a set \(w = \{\langle \text{add}, p_1 \rangle, \langle \text{remove}, p_2 \rangle, \langle \text{add}, p_3 \rangle\}\), \(w.\text{join} = \{p_1, p_2\}\), \(w.\text{remove} = \{p_1\}\), and \(w.\text{membership} = \{p_2\}\). For a time \(t\) in a run \(r\), we denote by \(V(t)\) the union of all sets \(q\) s.t. \(\text{reconfig}(q)\) completes before time \(t\) in \(r\). A configuration is a finite set of processes, and the current configuration at time \(t\) is \(V(t).\text{membership}\). We assume that only processes in \(V(t).\text{membership}\) invoke operations at time \(t\). The initial set of processes \(\Pi_0 \subset \Pi\) is known to all and we say, by convention, that \(\text{reconfig}(\{\langle \text{add}, p \rangle | p \in \Pi_0\})\) completes at time 0, i.e., \(V(0).\text{membership} = \Pi_0\).

We define \(P(t)\) to be the set of pending changes at time \(t\) in run \(r\), i.e., the set of all changes included in pending reconfig operations. We denote by \(F(t)\) the set of processes that have failed before time \(t\) in \(r\); initially, \(F(0) = \{\}\). For a series of arbitrary sets \(S(t)\), \(t \in \mathbb{N}\), we define \(S(*) \triangleq \bigcup_{t \in \mathbb{N}} S(t)\). The notation is illustrated in Figure 4.1.

**Correct processes and fairness** A process \(p\) is correct if \(p \in V(*)\). join \(\setminus F(*)\). A run \(r\) is fair if every send action by a correct process that is enabled infinitely often eventually occurs, and every message sent by a correct process \(p_i\) to a correct process \(p_j\) is eventually received at \(p_j\). Note that messages sent to a faulty process from a correct one may or may not be received. A process \(p\) is active if \(p\) is correct, and \(p \notin P(*)\). remove.

**Service specification** A linearization of a run \(r\) is an equivalent sequential run that preserves \(r\)’s operation precedence relation and the service’s sequential specification. The
sequential specification for a register is as follows: A read returns the latest written value, or \( v_0 \) if none was written. An MWMR register is atomic, also called linearizable [45], if every run has a linearization. Lamport [52] defines a safe single-writer register. Here, we generalize the definition to multi-writer registers in a weak way in order to strengthen the impossibility result. Intuitively, if a read is not concurrent with any write we require it to return a value that reflects some possible outcome of the writes that precede it; otherwise we allow it to return an arbitrary value. Formally: An MWMR register is safe if for every run \( r \) for every read operation \( rd \) that has no concurrent writes in \( r \), there is a linearization of the subsequence of \( r \) consisting of \( rd \) and the write operations in \( r \).

A wait-free service guarantees that every active process’s operation completes regardless of the actions of other processes.

**Failure model and reconfiguration** The reconfig operations determine which processes are allowed to fail at any given time. Static storage algorithms [10] tolerate failures of a minority of their (static) universe. At a time \( t \) when no reconfig operations are ongoing, the dynamic failure condition may be simply defined to allow less than \( \left| V(t) \text{membership} \right| / 2 \) failures of processes in \( V(t) \text{membership} \). When there are pending additions and removals, the rule must be generalized to take them into account. For our algorithm in Section 4.3, we adopt a generalization presented in previous works [8, 50, 72, 7]:

**Definition 16** (minority failures). A model allows minority failures if at all times \( t \) in \( r \), fewer than \( \left| V(t) \text{membership} \right| \setminus \left| P(t) \text{remove} \right| / 2 \) processes out of \( V(t) \text{membership} \cup P(t) \text{join} \) are in \( F(t) \).

Note that this failure condition allows processes whose remove operations have completed to be (immediately) safely switched off as it only restricts failures out of the current membership and pending joins. We say that a service is reconfigurable if failures of processes in \( V(t) \text{remove} \) are unrestricted.

In order to strengthen our lower bound in Section 4.2 we weaken the failure model. Like FLP [36], our lower bound applies as long as at least one process can fail. Formally, a failure is allowed whenever all failed processes have been removed and the current membership consists of at least three processes\(^1\). We call such a state “clean”, captured by the following predicate: \( \text{clean}(t) \triangleq (V(t) \text{membership} \cup P(t) \text{join}) \cap F(t) = \{ \} \land \left| V(t) \text{membership} \setminus P(t) \text{remove} \right| \geq 3 \). The minimal failure condition is thus defined as follows:

**Definition 17** (minimal failure). A model allows minimal failure if in every run \( r \) ending at time \( t \) when \( \text{clean}(t) \), for every process \( p \in V(t) \text{membership} \cup P(t) \), there is an extension of \( r \) where \( p \) fails at time \( t + 1 \).

Notice that the minority failure condition allows minimal failure, and so all algorithms that assume minority failures [8, 50, 72, 7] are a fortiori subject to our lower bound, which is proven for minimal failures.

\(^1\)Note that with fewer than three processes, even static systems cannot tolerate failures [10].
4.2 Impossibility of Wait-Free Dynamic Safe Storage

In this section we prove that there is no implementation of wait-free dynamic safe storage in a model that allows minimal failures. We construct a fair run with infinitely many reconfiguration operations in which a slow process \( p \) never completes its write operation. We do so by delaying all of \( p \)'s messages. A message from \( p \) to a process \( p_i \) is delayed until \( p_i \) is removed, and we make sure that all processes except \( p \) are eventually removed and replaced.

**Theorem 8.** There is no algorithm that emulates wait-free dynamic safe storage in an asynchronous system allowing minimal failures.

**Proof (Theorem 8).** Assume by contradiction that such an algorithm \( A \) exists. We prove two lemmas about \( A \).

**Lemma 30.** Consider a run \( r \) of \( A \) ending at time \( t \) s.t. \( \text{clean}(t) \) and two processes \( p_i, p_j \in V(t) \). Extend \( r \) by having \( p_j \) invoke operation \( op \) at time \( t + 1 \). Then there exists an extension of \( r \) where (1) \( op \) completes at some time \( t' > t \), (2) no process receives a message from \( p_i \) between \( t \) and \( t' \), and (3) no process fails and no operations are invoked between \( t \) and \( t' \).

**Proof:** To prove this lemma, we need to show that there exists a fair run \( \sigma_1 \) of \( A \) where \( p_i \) fails at time \( t + 2 \) and all of its in-transit messages are lost, no other process fails, and no operations are invoked. By wait-freedom, \( op \) eventually completes at some time \( t_1 \) in \( \sigma_1 \). Since \( p_i \) fails and all its outstanding messages are lost, then from time \( t \) to \( t_1 \) no process receives any messages from \( p_i \). Now let \( \sigma_2 \) be identical to \( \sigma_1 \) except that \( p_i \) does not fail, but all of its messages are delayed. Note that \( \sigma_1 \) and \( \sigma_2 \) are indistinguishable to all processes except \( p_i \). Thus, \( op \) returns at time \( t_1 \) also in \( \sigma_2 \).

**Lemma 31.** Consider a run \( r \) of \( A \) ending at time \( t \) s.t. \( \text{clean}(t) \). Let \( v_1 \in V \setminus \{v_0\} \) be a value s.t. no process invokes \( \text{write}(v_1) \) in \( r \). If we extend \( r \) fairly so that \( p_i \) invokes \( w = \text{write}(v_1) \) at time \( t + 1 \) which completes at some time \( t_1 > t + 1 \) s.t. \( \text{clean}(t_1) \) for all \( t < t' \leq t_1 \) then in the run fragment between \( t + 1 \) and \( t_1 \), some process \( p_k = p_i \) receives a message sent by \( p_i \).

**Proof:** Assume by way of contradiction that in the run fragment between \( t + 1 \) and \( t_1 \) no process \( p_k = p_i \) receives a message sent by \( p_i \), and consider a run \( r' \) that is identical to \( r \) until time \( t_1 \) except that \( p_i \) does not invoke \( w \) at time \( t \). Now assume that some process \( p_j = p_i \) invokes a \( \text{read} \) operation at time \( t_1 + 1 \) in \( r' \). By the assumption, \( \text{clean}(t_1) \) and therefore \( \text{clean}(t_1 + 1) \). Thus, by Lemma 30, there is a run fragment \( r' \) beginning at the final state of \( r' \) (time \( t_1 + 1 \)), where \( \text{rd} \) completes at some time \( t_2 \) s.t. between \( t_1 + 1 \) and \( t_2 \) no process receives a message from \( p_i \). Since
no process invokes write($v_1$) in $r'$, and no writes are concurrent with the read, by safety, $rd$ returns some $v_2 \neq v_1$.

Now notice that all global states from time $t$ to time $t_1$ in $r$ and $r'$ are indistinguishable to all processes except $p_1$. Thus, we can continue run $r$ with an invocation of read operation $rd'$ by $p_1$ at time $t_1$, and append $\sigma$ to it. Operation $rd'$ hence completes and returns $v_2$. A contradiction to safety.

To prove the theorem, we construct an infinite fair run $r$ in which a write operation of an active process never completes, in contradiction to wait-freedom.

Consider some initial global state $c_0$, s.t. $P(0) = F(0) = \{\}$ and $V(0).membership = \{p_1 \ldots p_n\}$, where $n \geq 3$. An illustration of the run for $n = 4$ is presented in Figure 4.2. Now, let process $p_1$ invoke a write operation $w$ at time $t_1 = 0$, and do the following:

Let process $p_n$ invoke reconfig($q$) where $q = \{\langle add, p_j\rangle | n + 1 \leq j \leq 2n - 2\}$ at time $t_1$. The state at the end of $r$ is clean (i.e., clean($t_1$)). So by Lemma 30, we can extend $r$ with a run fragment $\sigma_1$ ending at some time $t_2$ when reconfig($q$) completes, where no process $p_j \neq p_1$ receives a message from $p_1$ in $\sigma_1$, no other operations are invoked, and no process fails.

Then, at time $t_2 + 1$, $p_n$ invokes reconfig($q'$), where $q' = \{\langle remove, p_j\rangle | 2 \leq j \leq n - 1\}$. Again, the state is clean and thus by Lemma 30 again, we can extend $r$ with a run fragment $\sigma_2$ ending at some time $t_3$ when reconfig($q'$) completes s.t. no process $p_j \neq p_1$ receives a message from $p_1$ in $\sigma_2$, no other operations are invoked, and no process fails.

Recall that the minimal failures condition satisfies reconfigurability, i.e., all the processes in $V(t_3).remove$ can be in $F(t_3)$ (fail). Let the processes in $\{p_j | 2 \leq j \leq n - 1\}$ fail at time $t_3$, and notice that the fairness condition does not mandate that they receive messages from $p_1$. Next, allow $p_1$ to perform all its enabled actions till some time $t_4$.

Now notice that at $t_4$, $|V(t_4).membership| = n$, $P(t_4) = \{\}$, $(V(t_4).membership \cup P(t_4).join) \cap F(t_4) = \{\}$, and $|V(t_4).membership \setminus P(t_4).removal| \geq 3$. We can rename the processes in $V(t_4).membership$ (except $p_1$) so that the process that performed the remove and add operations becomes $p_2$, and all others get names in the range $p_3 \ldots p_n$. We can then repeat the construction above. By doing so infinitely many times, we get an infinite run $r$ in which $p_1$ is active and no process ever receives a message from $p_1$. However, all of $p_1$’s enabled actions eventually occur. Since no process except $p_1$ is correct in $r$, the run is fair. In addition, since clean($t$) for all $t$ in $r$, by the contrapositive of Lemma 31, $w$ does not complete in $r$, and we get a violation of wait-freedom.

4.3 Oracle-Based Dynamic Atomic Storage

We present an algorithm that circumvents the impossibility result of Section 4.2 using a failure detector. In this section we assume the minority failure condition. In Section 4.3.1,
we define a dynamic eventually perfect failure detector. In Section 4.3.2, we describe an algorithm, based on dynamic state machine replication, that uses the failure detector to implement a wait-free dynamic atomic MWMR register. The algorithm’s correctness is proven in Section 4.3.3.

### 4.3.1 Dynamic failure detector

Since the set of processes is potentially infinite, we cannot have the failure detector report the status of all processes as static failure detectors typically do. Dynamic failure detectors addressing this issue have been defined in previous works, either providing a set of processes that have been excluded from or included into the group [54], or assuming that there is eventually a fixed set of participating processes [28]. In our model, we do not assume that there is eventually a fixed set of participating processes, as the number of reconfig operations can be infinite. And we do not want the failure detector to answer with a list of processes, because in dynamic systems, this gives additional information about participating processes that could have been unknown to the inquiring process, and thus it is not clear how such a failure detector can be implemented.

Instead, our dynamic failure detector is queried separately about each process. For each query, it answers either fail or ok. It can be wrong for an unbounded period, but for each process, it eventually returns a correct answer. Formally, a dynamic eventually perfect failure detector, $\diamond P^D$, satisfies two properties:

- **Strong completeness:** For each process $p_i$ that fails at time $t_i$, there is a time $t > t_i$ s.t. the failure detector answers fail to every query about $p_i$ after time $t$.

- **Eventual strong accuracy:** There exists a time $t$, called the stabilization time, s.t. the failure detector answers ok to every query at any time $t' > t$ about a correct process in $V(t'), join$.

Note that $\diamond P^D$ can be implemented in a standard way in the eventually (partially) synchronous model by pinging the queried process and waiting for a response until a timeout.
4.3.2 Dynamic storage algorithm

We first give the overview of our algorithm and then present the full description.

Algorithm overview

The key to achieving liveness with unbounded reconfig operations is a novel helping mechanism, which is based on our failure detector. Intuitively, the idea is that every process tries to help all other processes it believes are correct (according to its failure detector), to complete their concurrent operations together with its own. At the beginning of an operation, a process \( p \) queries all other processes it knows about for the operations they currently perform. The failure detector is needed in order to make sure that (1) \( p \) does not wait forever for a reply from a faulty process (achieved by strong completeness), and (2) every slow correct process eventually gets help (achieved by eventual strong accuracy).

State machine emulation of a register  We use a state machine \( sm \) to emulate a wait-free atomic dynamic register, DynaReg. Every process has a local replica of \( sm \), and we use consensus to agree on \( sm \)'s state transitions. Notice that each process is equipped with a failure detector FD of class \( \mathcal{D}^D \), so consensus is solvable under the assumption of a correct majority in a given configuration [53].

Each instance of consensus runs in some static configuration \( c \) and is associated with a unique timestamp. A process participates in a consensus instance by invoking a propose operation with the appropriate configuration and timestamp, as well as its proposed decision value. Consensus then responds with a decide event, so that the following properties are satisfied: Uniform Agreement – every two decisions are the same. Validity – every decision was previously proposed by one of the processes in \( c \). Termination – if a majority of \( c \) is correct, then eventually every correct process in \( c \) decides. We further assume that a consensus instance does not decide until a majority of the members of the configuration propose in it.

The \( sm \) (lines 2-5 in Algorithm 7) keeps track of dynaReg's value in a variable \( val \), and the configuration in a variable \( cng \), containing both a list of processes, \( cng.mem \), and a set of removed processes, \( cng.rem \). Write operations change \( val \), and reconfig operations change \( cng \). A consensus decision may bundle a number of operations to execute as a single state transition of \( sm \). The number of state transitions executed by \( sm \) is stored in the variable \( ts \). Finally, the array \( lastOps \) maps every process \( p \) in \( cng.mem \) to the sequence number (based on \( p \)'s local count) of \( p \)'s last operation that was performed on the emulated DynaReg together with its result.

Each process partakes in at most one consensus at a time; this consensus is associated with timestamp \( sm.ts \) and runs in \( sm.cng.mem \). In every consensus, up to \( |sm.cng.mem| \) ordered operations on the emulated DynaReg are agreed upon, and \( sm \)'s state changes according to the agreed operations. A process's \( sm \) may change either when consensus
decides or when the process receives a newer \( sm \) from another process, in which case it skips forward. So \( sm \) goes through the same states in all the processes, except when skipping forward. Thus, for every two processes \( p_k, p_l \), if \( sm_k.ts = sm_l.ts \), then \( sm_k = sm_l \). (A subscript \( i \) indicates the variable is of process \( p_i \).)

Helping

The problematic scenario in the impossibility proof of Section 4.2 occurs because of endless reconfig operations, where a slow process is never able to communicate with members of its configuration before they are removed. In order to circumvent this problem, we use FD to implement a helping mechanism. When proposing an op-
Chapter 4. On Liveness of Dynamic Storage

operation, process $p_i$ tries to help other processes in two ways: first, it helps them complete operations they may have successfully proposed in previous rounds (consensuses) but have not learned about their outcomes; and second, it proposes their new operations. To achieve the first, it sends a helping request with its $sm$ to all other processes in $sm_i$.cng.mem. For the second, it waits for each process to reply with a help reply containing its latest invoked operation, and then proposes all the operations together. Processes may fail or be removed, so $p_i$ cannot wait for answers forever. To this end, we use FD. For every process in $sm_i$.cng.mem that has not been removed, $p_i$ repeatedly inquires FD and waits either for a reply from the process or for an answer from FD that the process has failed. Notice that the strong completeness property guarantees that $p_i$ will eventually continue, and strong accuracy guarantees that every slow active process will eventually receive help in case of endless reconfig operations.

Nevertheless, if the number of reconfig operations is finite, it may be the case that some slow process is not familiar with any of the correct members in the current configuration, and no other process performs an operation (hence, no process is helping). To ensure progress in such cases, every correct process periodically sends its $sm$ to all processes in its $sm_i$.cng.mem.

**State survival**  Before the reconfig operation can complete, the new $sm$ needs to propagate to a majority of the new configuration, in order to ensure its survival. Therefore, after executing the state transition, $p_i$ sends $sm_i$ to $sm_i$.cng members and waits until it either receives acknowledgements from a majority or learns of a newer $sm$. Notice that in the latter case, consensus in $sm_i$.cng.mem has decided, meaning that at least a majority of $sm_i$.cng.mem has participated in it, and so have learned of it.

**Flow example**  The algorithm flow is illustrated in Figure 4.3. In this example, a slow process $p_2$ invokes operation $op_{21}$ before FD’s stabilization time, $ST$. Process $p_1$ invokes operation $op_{11} = \langle \text{add}, p_3 \rangle$ after $ST$. It first sends $helpRequest$ to $p_2$ and waits for it to reply with $helpReply$. Then it proposes $op_{21}$ and $op_{11}$ in a consensus. When $decide$ occurs, $p_1$ updates its $sm$, sends it to all processes, and waits for majority. Then $op_{11}$ returns and $p_1$ fails before $p_2$ receives its update message. Next, $p_3$ invokes a reconfig operation, but this time when $p_2$ receives $helpRequest$ with the up-to-date $sm$ from $p_3$, it notices that its operation has been performed, and $op_{21}$ returns.

**Detailed description**

The data structure of process $p_i$ is given in Algorithm 7. The type $Ops$ defines the representation of operations. The emulated state machine, $sm_i$, is described above. Integer $opNum_i$ holds the sequence number of $p_i$’s current operation; $ops_i$ is a set that contains operations that need to be completed for helping; the flag $pend_i$ is a boolean that indicates whether or not $p_i$ is participating in an ongoing consensus; and $myOp_i$ is the latest operation invoked at $p_i$. 
The algorithm of process $p_i$ is presented in Algorithms 8 and 9. We execute every event handler, (operation invocation, message receiving, and consensus decision) atomically excluding wait instructions; that is, other event handlers may run after the handler completes or during a wait (lines 16,18,27 in Algorithm 8). The algorithm runs in two phases. The first, *gather*, is described in Algorithm 8 lines 11–16 and in Algorithm 9 lines 52–58. Process $p_i$ first increases its operation number $opNum_i$, writes $op$ together with $opNum_i$ to the set of operations $ops_i$, and sets $myOp_i$ to be $op$. Then it sends ("helpRequest", . . .) to every member of $A = sm_i.cng.mem$ (line 15), and waits for each process in $A$ that is not suspected by the FD or removed to reply with ("helpReply", . . .). Notice that $sm_i$ may change during the wait because messages are handled, and $p_i$ may learn of processes that have been removed.

When ("helpRequest", $num$, $sm$) is received by process $p_i \neq p_i$, if the received $sm$ is newer than $sm_i$, then process $p_i$ adopts $sm$ and abandons any previous consensus. Either way, $p_i$ sends ("helpReply", . . .) with its current operation $myOp_i$ in return.

Upon receiving ("helpReply", $opNum_i$, $op$, $num$) that corresponds to the current operation number $opNum_i$, process $p_i$ adds the received operation $op$, its number $num$, and the identity of the sender to the set $ops_i$.

At the end of this phase, process $p_i$ holds a set of operations, including its own, that it tries to agree on in the second phase (the order among this set is chosen deterministically, as explained below). Note that $p_i$ can participate in at most one consensus per timestamp, and its propose might end up not being the decided one, in which case it may need to propose the same operations again. Process $p_i$ completes $op$ when it discovers that $op$ has been performed in $sm_i$, whether by itself or by another process.

The second phase appears in Algorithm 8 lines 17–28, and in Algorithm 9 lines 31–51. In line 17, $p_i$ checks if its operation has not been completed yet. In line 18, it waits until it does not participate in any ongoing consensus ($pend_i=false$) or some other process helps it complete $op$. Recall that during a wait, other events can be handled. So if a message with an up-to-date $sm$ is received during the wait, $p_i$ adopts the $sm$. In case $op$ has been completed in $sm$, $p_i$ exits the main while (line 19). Otherwise, $p_i$ waits until it does not participate in any ongoing consensus. This can be the case if (1) $p_i$ has not proposed yet, (2) a message with a newer $sm$ was received and a previous consensus was subsequently abandoned, or (3) a *decide* event has been handled. In all cases, $p_i$ marks that it now
participates in consensus in line 20, prepares a new request \( \text{Req} \) with the operations in \( \text{ops}_i \), that have not been performed yet in \( \text{sm}_i \) in line 27, proposes \( \text{Req} \) in the consensus associated with \( \text{sm}_i,\text{ts} \), and sends \( \langle \text{"propose"}, \ldots \rangle \) to all the members of \( \text{sm}_i,\text{cng}.\text{mem} \).

**Algorithm 8** Process \( p_i \)'s algorithm: performing operations

10: upon invoke operation\( (\text{op}) \) do
11: \( \text{opNum}_i \leftarrow \text{opNum}_i + 1 \)
12: \( \text{ops}_i \leftarrow \{ \langle p_i, \text{op}, \text{opNum}_i \rangle \} \)
13: \( \text{myOp}_i \leftarrow \text{op} \)
14: \( A \leftarrow \text{sm}_i,\text{cng}.\text{mem} \)
15: for all \( p \in A \) send \( \langle \text{"helpRequest"}, \text{opNum}_i, \text{sm}_i \rangle \) to \( p \)
16: for all \( p \in A \) wait for \( \langle \text{"helpReply"}, \text{opNum}_i, \ldots \rangle \) from \( p \) or \( p \) is suspected or \( p \in \text{sm}_i,\text{cng}.\text{mem} \)
17: while \( \text{sm}_i,\text{lastOps}[i].\text{num} \neq \text{opNum}_i \)
18: \( \text{wait until } \neg \text{pend}_i \) or \( \text{sm}_i,\text{lastOps}[i].\text{num} = \text{opNum}_i \)
19: if \( \text{sm}_i,\text{lastOps}[i].\text{num} = \text{opNum}_i \) then break
20: \( \text{pend}_i \leftarrow \text{true} \)
21: \( \text{Req} \leftarrow \{ \langle p_j, \text{op}, \text{num} \rangle \mid \text{num} > \text{sm}_j,\text{lastOps}[j].\text{num} \} \)
22: propose\( (\text{sm}_i,\text{cng},\text{sm}_i,\text{ts}, \text{Req}) \)
23: for all \( p \in \text{sm}_i,\text{cng}.\text{mem} \) send \( \langle \text{"propose"}, \text{sm}_i, \text{Req} \rangle \) to \( p \)
24: if \( \text{op}.\text{type} = \text{REC} \)
25: \( \text{ts} \leftarrow \text{sm}_i,\text{ts} \)
26: for all \( p \in \text{sm}_i,\text{cng}.\text{mem} \) send \( \langle \text{"update"}, \text{sm}_i, \text{opNum}_i \rangle \) to \( p \)
27: \( \text{wait for } \langle \text{"ACK"}, \text{opNum}_i \rangle \) from \( p \) and \( \text{sm}_i,\text{ts} > \text{ts} \)
28: return \( \text{sm}_i,\text{lastOps}[i].\text{res} \)
29: periodically,
30: for all \( p \in \text{sm}_i,\text{cng}.\text{mem} \) send \( \langle \text{"update"}, \text{sm}_i, \bot \rangle \) to \( p \)

When \( \langle \text{"propose"}, \text{sm}, \text{Req} \ldots \rangle \) is received by process \( p_j \neq p_i \), if the received \( \text{sm} \) is more updated than \( \text{sm}_i \), then process \( p_j \) adopts \( \text{sm} \), abandons any previous consensus, proposes \( \text{Req} \) in the consensus associated with \( \text{sm}_i,\text{ts} \), and forwards the message to all other members of \( \text{sm}_i,\text{cng}.\text{mem} \). The same is done if \( \text{sm} \) is identical to \( \text{sm}_i \) and \( p_j \) has not proposed yet in the consensus associated with \( \text{sm}_i,\text{ts} \). Otherwise, \( p_j \) ignores the message.

The event \( \text{decide}_i(\text{sm},\text{cng},\text{sm}_i,\text{ts}, \text{Req}) \) indicates a decision in the consensus associated with \( \text{sm}_i,\text{ts} \). When this occurs, \( p_i \) performs all the operations in \( \text{Req} \) and changes \( \text{sm}_i \)'s state. It sets the value of the emulated DynaReg, \( \text{sm}_i,\text{value} \), to be the value of the write operation of the process with the lowest id, and updates \( \text{sm}_i,\text{cng} \) according to the \text{reconfig} operations. In addition, for every \( \langle p_j, \text{op}, \text{num} \rangle \in \text{Req} \), \( p_i \) writes to \( \text{sm}_i,\text{lastOps}[j] \), \( \text{num} \) and \( \text{op} \)'s response, which is “ok” in case of a write or a \text{reconfig}, and \( \text{sm}_i,\text{value} \) in case of a read. Next, \( p_i \) increases \( \text{sm}_i,\text{ts} \) and sets \( \text{pend}_i \) to false, indicating that it no longer participates in any ongoing consensus.

Finally, after \( \text{op} \) is performed, \( p_i \) exits the main while. If \( \text{op} \) is not a \text{reconfig} operation, then \( p_i \) returns the result, which is stored in \( \text{sm}_i,\text{lastOps}[i].\text{res} \). Otherwise, before returning, \( p_i \) has to be sure that a majority of \( \text{sm}_i,\text{cng}.\text{mem} \) receives \( \text{sm}_i \). It sends \( \langle \text{"update"}, \text{sm}_i \ldots \rangle \) to all the processes in \( \text{sm}_i,\text{cng}.\text{mem} \) and waits for \( \langle \text{"ACK"}, \ldots \rangle \) from a majority of them. Notice that it may be the case that there is no such correct majority due to later \text{reconfig} operations and failures, so, \( p_i \) stops waiting when a more updated \( \text{sm} \) is
invoked from some point on, every correct process periodically sends ⟨p⟩ to all processes in its sm. This implies that a majority of sm, cng, mem has already received sm, i (since a majority is needed in order to solve consensus).

Upon receiving ⟨“update”, sm, num⟩ with a new sm from process p, process pj adopts sm and abandons any previous consensus. In addition, if num ≠ ⊥, pj sends ⟨“ACK”, num⟩ to pi (Algorithm 9 lines 59–63).

Beyond handling operations, in order to ensure progress in case no operations are invoked from some point on, every correct process periodically sends ⟨“update”, sm, ⊥⟩ to all processes in its sm, cng, mem (Algorithm 8 line 30).

---

**Algorithm 9** Process p’s algorithm: event handlers

```plaintext
31: upon decide(sm, cng, sm, ts, Req) do
32: W ← \{ (p, value) | (p, (WR, value), num) ∈ Req \}
33: if W ≠ {} then ▷ deterministically choose one of the writes to be the last
34: sm, value ← value with smallest p in W
35: for all (p, op, num) ∈ Req do ▷ apply op to sm
36: if op.type = WR then
37: sm, cng, rem ← sm, cng, rem ∪ { p | remove, p } ∈ op, changes
38: sm, cng, mem ← sm, cng, mem ∪ { p | add, p } ∈ op, changes \ sm, cng, rem
39: sm, lastOps[] ← ⟨ num, sm, value ⟩
40: else
41: sm, cng, rem ← sm, cng, rem ∪ { p | ⟨ remove, p ⟩ ∈ op, changes } ⊖ sm, cng, rem
42: sm, cng, mem ← sm, cng, mem ∪ { p | ⟨ add, p ⟩ ∈ op, changes } \ sm, cng, rem
43: sm, lastOps[] ← ⟨ num, “ok” ⟩
44: sm, ts ← sm, ts + 1
45: pend, ← false
46: upon receiving ⟨“propose”, sm, Req⟩ from pj do
47: if (sm, ts > sm, ts) or (sm, ts = sm, ts ∧ pend, = true) then return
48: sm, ← sm
49: pend, ← true
50: propose(sm, cng, sm, ts, Req)
51: for all p ∈ sm, cng, mem send ⟨“propose”, sm, Req⟩ to p
52: upon receiving ⟨“helpRequest”, num, sm⟩ from pj do
53: if sm, ts < sm, ts then ▷ learn new sm
54: sm, ← sm
55: pend, ← false
56: send ⟨“helpReply”, num, myOp, opNum⟩
57: upon receiving ⟨“helpReply”, opNum, op, num⟩ from pj do
58: ops, ← ops, ∪ { p, op, num }
59: upon receiving ⟨“update”, sm, num⟩ from pj do
60: if sm, ts < sm, ts then ▷ learn new sm
61: sm, ← sm
62: pend, ← false
63: if num ≠ ⊥ then send ⟨“ACK”, num⟩ to pi
```
4.3.3 Correctness proof

Atomicity

Every operation is uniquely defined by the process that invoked it and its local number. During the proof we refer to operation \( op \) invoked by process \( p_i \) with local number \( opNum_i = n \) as the tuple \( \langle p_i, op, n \rangle \). We begin the proof with three lemmas that link completed operations to \( sm \) states.

Lemma 32. Consider operation \( op \) invoked by some process \( p_i \) in \( r \) with local number \( opNum_i = n \). If \( op \) returns in \( r \) at time \( t \), then there is at least one request \( Req \) that contains \( \langle p_i, op, n \rangle \) and has been chosen in a consensus in \( r \) before time \( t \).

Proof. When operation \( op \) return, \( sm_i.lastOps[i].num = n \) (line 17 or 18 in Algorithm 8). Processes update \( sm \) during a decide handler, or when a newer \( sm \) is received, and the first update occurs when some process \( p_j \) writes \( n \) to \( sm_j.lastOps[i].num \) during a decide handler. In the decide handler, \( n \) is written to \( sm.lastOps[i].num \) when the chosen request in the corresponding consensus contains \( \langle p_i, op, n \rangle \).

Lemma 33. For two processes \( p_i, p_j \), let \( t \) be a time in a run \( r \) in which neither \( p_i \) or \( p_j \) is executing a decide handler. Then at time \( t \), if \( sm_i.ts = sm_j.ts \), then \( sm_i = sm_j \).

Proof. We prove by induction on timestamps. Initially, all correct processes have the same \( sm \) with timestamp 0. Now consider timestamp \( TS \), and assume that for every two processes \( p_i, p_j \) at any time not during the execution of decide handlers, if \( sm_i.ts = sm_j.ts = TS \), then \( sm_i = sm_j \). Processes increase their \( sm.ts \) to \( TS + 1 \) either at the end of a decide handler associated with \( TS \) or when they receive a message with \( sm \) s.t. \( sm.ts = TS + 1 \). By the agreement property of consensus and by the determinism of the algorithm, all the processes that perform the decide handler associated with \( TS \) perform the same operations, and therefore move \( sm \) (at the end of the handler) to the same state. It is easy to show by induction that all the processes that receive a message with \( sm \) s.t. \( sm.ts = TS + 1 \) receive the same \( sm \). The lemma follows.

Observation 10. For two process \( p_i, p_j \), let \( sm_1 \) and \( sm_2 \) be the values of \( sm \) at two different times in a run \( r \). If \( sm_1.ts \geq sm_2.ts \), then \( sm_1.lastOps[i].num \geq sm_2.lastOps[i].num \).

Lemma 34. Consider operation \( \langle p_i, op, opNum_i \rangle \) invoked in \( r \) with \( opNum_i = n \). Then \( \langle p_i, op, n \rangle \) is part of at most one request that is chosen in a consensus in \( r \).

Proof. Assume by way of contradiction that \( \langle p_i, op, n \rangle \) is part of more than one request that is chosen in a consensus in \( r \). Now consider the earliest one, \( Req \), and assume that it is chosen in a consensus associated with timestamp \( TS \). At the end of the decide handler associated with timestamp \( TS \), \( sm.lastOps[i].num = n \) and the timestamp is increased to \( TS + 1 \). Thus, by Lemma 33 \( sm.lastOps[i].num = n \) holds for every \( sm \) s.t. \( sm.ts = TS + 1 \).
TS + 1. Consider now the next request, Req1, that contains \( \langle p_i, op, n \rangle \), and is chosen in a consensus. Assume that this consensus associated with timestamp \( TS' \), and notice that \( TS' > TS \). By the validity of consensus, this request is proposed by some process \( p_j \), when \( sm_j.ts \) is equal to \( TS' \). By Observation 10, at this point \( sm_j.lastOps[i].num \geq n \), and therefore \( p_j \) does not include \( \langle p_i, op, n \rangle \) in Req1 (line 27 in Algorithm 8). A contradiction.

Based on the above lemmas, we can define, for each run \( r \), a linearization \( \sigma_r \), where operations are ordered as they are chosen for execution on \( sm's \) in \( r \).

**Definition 18.** For a run \( r \), we define the sequential run \( \sigma_r \) to be the sequence of operations decided in consensus instances in \( r \), ordered by the order of the chosen requests they are part of in \( r \). The order among operations that are part of the same chosen request is the following: first all writes, then all reads, and finally, all reconfig operations. Among each type, operations are ordered by the process ids of the processes that invoked them, from the highest to the lowest.

Note that for every run \( r \), the sequential run \( \sigma_r \) is well defined. Moreover, \( \sigma_r \) contains every completed operation in \( r \) exactly once, and every invoked operation at most once.

In order to prove atomicity we show that (1) \( \sigma_r \) preserves \( r \)'s real time order (lemma 35); and (2) every read operation \( rd \) in \( r \) returns the value that was written by the last write operation that precedes \( rd \) in \( \sigma_r \), or \( \perp \) if there is no such operation (lemma 36).

**Lemma 35.** If operation \( op_1 \) returns before operation \( op_2 \) is invoked in \( r \), then \( op_1 \) appears before \( op_2 \) in \( \sigma_r \).

**Proof.** By Lemma 32, \( op_1 \) is part of a request \( Req_1 \) that is chosen in a consensus before \( op_2 \) is invoked, and thus \( op_2 \) cannot be part of \( Req_1 \) or any other request that is chosen before \( Req_1 \). Hence \( op_1 \) appears before \( op_2 \) in \( \sigma_r \).

**Lemma 36.** Consider read operation \( rd = \langle p_i, RD, n \rangle \) in \( r \), which returns a value \( v \). Then \( v \) is written by the last write operation that precedes \( rd \) in \( \sigma_r \), or \( v = \perp \) if there is no such operation.

**Proof.** By Lemmas 32 and 34, \( rd \) is part of exactly one request \( Req_1 \) that is chosen in a consensus, associated with some timestamp \( TS \). Thus \( sm.lastOps[i] \) is set to \( \langle n, val \rangle \) in the decide handler associated with \( TS \). By Lemma 33, \( sm.lastOps[i] = \langle n, val \rangle \) for all \( sm \) s.t. \( sm.ts = TS + 1 \). By Lemma 34 and since we consider only well-formed runs, \( sm.lastOps[i] = \langle n, val \rangle \) when \( rd \) returns, and therefore \( rd \) returns \( val \). Now consider three cases:

- There is no write operation in \( Req_1 \) or in any request that was chosen before \( Req_1 \) in \( r \). In this case, there is no write operation before \( rd \) in \( \sigma_r \), and no process writes to \( sm.value \) before \( sm.lastOps[i] \) is set to \( \langle n, val \rangle \), and therefore, \( rd \) returns \( \perp \) as expected.
Chapter 4. On Liveness of Dynamic Storage

- There is a write operation in Req₁ in r. Consider the write operation w in Req₁ that is invoked by the process with the lowest id, and assume its argument is v'. Notice that w is the last write that precedes rd in σᵣ. By the code of the decide handler, sm.value equals v' at the time when sm.lastOps[i] is set to ⟨n, val⟩. Therefore, val = v', rd returns the value that is written by the last write operation that precedes it in σᵣ.

- There is no write operation in Req₁, but there is a request that contains a write operation and is chosen before Req₁ in r. Consider the last such request Req₂, and consider the write operation w invoked by the process with the lowest id in Req₂. Assume that w's argument is v', and Req₂ was chosen in a consensus associated with timestamp TS' (notice that TS' < TS). By the code of the decide handler and Lemma 33, in all the sm's s.t. sm.ts = TS' + 1, the value of sm.value is v'. Now, since there is no write operation in any chosen request between Req₂ and Req₁ in r, no process writes to sm.value when TS' < sm.ts < TS. Hence, when sm.lastOps[i] is set to ⟨n, val⟩, sm.value equals v', and thus val = v'. Therefore, rd returns the value that is written by the last write operation that precedes rd in σᵣ. □

Corollary 12. Algorithms 7–9 implement an atomic storage service.

Liveness

Consider operation opᵢ invoked at time t by a correct process pᵢ in run r. Notice that r is a run with either infinitely or finitely many invocations. We show that, in both cases, if pᵢ is active in r, then opᵢ returns in r.

We associate the addition or removal of process pᵢ by a process pᵢ with the timestamp that equals sm.ts at the time when the operation returns. The addition of all processes in P₀ is associated with timestamp 0.

First, we consider runs with infinitely many invocations. In Lemma 37, we show that for every process p, every sm associated with a larger timestamp than p's addition contains p in sm.cng.mem. In Observation 3, we show that in a run with infinitely many invocations, for every timestamp ts, there is a completed operation that has a bigger timestamp than ts at the time of the invocation. Moreover, after the stabilization time of the FD, operations must help all the slow active processes in order to complete. In Lemma 38, we use the observation to show that any operation invoked in a run with infinitely many invocations returns.

Next, we consider runs with finitely many invocations. We show Lemma 39 that eventually all the active members of the last sm adopt it. Then, in Lemma 40, we show that every operation invoked by an active process completes. Finally, Theorem 9, stipulates that the algorithm satisfies wait-freedom.
Lemma 37. Assume the addition of \( p_i \) is associated with timestamp \( TS \) in run \( r \). If \( p_i \) is active, then \( p_i \in sm.cng.mem \) for every \( sm \) s.t. \( sm.ts \geq TS \).

Proof. The proof is by induction on \( sm.ts \). Base: If \( p_i \in P_0 \), then \( p_i \in sm.cng.mem \) for all \( sm \) s.t. \( sm.ts = 0 \). Otherwise, \( \langle \text{add}, p_i \rangle \) is part of a request that is chosen in a consensus associated with timestamp \( TS' = TS - 1 \), and thus, by Lemma 33, \( p_i \in sm.cng.mem \) for all \( sm \) s.t. \( sm.ts = TS' + 1 = TS \). Induction: Process \( p_i \) is active, so no process invokes \( \langle \text{remove}, p_i \rangle \), and therefore, together with the validity of consensus, no chosen request contains \( \langle \text{remove}, p_i \rangle \). Hence, if \( p_i \in sm.cng.mem \) for \( sm \) with \( sm.ts = k \), then \( p_i \in sm.cng.mem \) for every \( sm \) s.t. \( sm.ts = k + 1 \).

\[ \square \]

Claim 3. Consider a run \( r \) of the algorithm with infinitely many invocations. Then for every time \( t \) and timestamp \( TS \), there is a completed operation that is invoked after time \( t \) by a process with \( sm.ts > TS \) at the time of the invocation.

Proof. Recall that \( r \) is well-formed and only processes in \( V(t).join \) can invoke operations at time \( t \). Therefore, there are infinitely many completed operations in \( r \). Since a finite number of operations are completed with each timestamp, the claim follows.

\[ \square \]

Lemma 38. Consider an operation \( op_i \) invoked at time \( t \) by an active process \( p_i \) in a run \( r \) with infinitely many invocations. Then \( op_i \) completes in \( r \).

Proof. Assume by way of contradiction that \( p_i \) is active and \( op_i \) does not complete in \( r \). Assume w.l.o.g. that \( p_i \)'s addition is associated with timestamps \( TS \) and \( op_i \) is invoked with \( opNum_i = n \). Consider a time \( t' > t \) after \( p_i \) invokes \( op_i \) and the FD has stabilized. By Claim 3, there is a completed operation \( op_j \) in \( r \), invoked by some process \( p_j \) at a time \( t'' > t' \) when \( sm_i.ts > TS \), whose completion is associated with timestamp \( TS' \).

By Lemma 37, \( p_i \in sm_i.cng.mem \), at time \( t'' \). Now by the algorithm and by the eventual strong accuracy property of the FD, \( p_j \) proposes \( op_j \) and \( op_i \) in the same request, and continues to propose both of them until one is selected. Note that it is impossible for \( op_j \) to be selected without \( op_i \) since any process that helps \( p_j \) after stabilization also helps \( p_i \). Hence, since \( op_j \) completes, they are both performed in the same decide handler. The run is well-formed, so \( p_i \) does not invoke operations that are associated with \( opNum_i > n \). Hence, following the time when \( op_i \) is selected, for all \( sm \) s.t. \( sm.ts > TS' \), \( sm.lastOps[i].num = n \). Now, again by Claim 3, consider a completed operation \( op_k \) in \( r \), that is invoked by some process \( p_k \) at time \( t''' \) after the stabilization time of the FD s.t. \( sm_k.ts > TS \) at time \( t''' \). Operation \( op_k \) cannot complete until \( p_i \) receives \( p_k \)'s \( sm \). Therefore, \( p_i \) receives \( sm \) s.t. \( sm.ts \geq TS' \), and thus \( sm.lastOps[i].num = n \). Therefore, \( p_i \) learns that \( op_i \) was performed, and \( op_i \) completes. A contradiction.

\[ \square \]

We now proceed to prove liveness in runs with finitely many invocations.
Definition 19. For every run \( r \) of the algorithm, and for any point \( t \) in \( r \), let \( T S_t \) be the timestamp associated with the last consensus that made a decision in \( r \) before time \( t \). Define \( s m^t \), at any point \( t \) in \( r \), to be the \( s m \)'s state after the completion of the decide handler associated with timestamp \( T S_t \) at any process. By Lemma 33, \( s m^t \) is unique. Recall that \( s m^0 \) is the initial state.

Claim 4. For every run \( r \) of the algorithm, and for any point \( t \) in \( r \), there is a majority of \( s m^t \).

Proof. By the code of the algorithm, for every run \( r \) and for any point \( t \) in \( r \), \( V(t).membership \subseteq s m^t \).cng.mem and \( s m^t \).cng.mem \( \cap V(t).remove = \{ \} \). The claim follows from failure condition.

Observation 11. Consider a run \( r \) of the algorithm with finitely many invocations. Then there is a point \( t \) in \( r \) s.t. for every \( t' > t \), \( s m^t = s m^{t'} \). Denote this \( s m \) to be \( s h n \).

The following lemma follows from Lemma 33, Claim 4, and the periodic update messages; for space limitations, we omit its proof.

Lemma 39. Consider a run \( r \) of the algorithm with finitely many invocations. Then eventually for every active process \( p_i \in s h n \).cng.mem, \( s m_i = s h n \).

Lemma 40. Consider an operation \( op_i \) invoked at time \( t \) by an active process \( p_i \) in a run \( r \) with finitely many invocations. Then \( op_i \) completes in \( r \).

Proof. By Lemma 37, \( p_i \in s h n \).cng.mem, and by Lemma 39, there is a point \( t' \) in \( r \) s.t. \( s m_i = s h n \) for all \( t \geq t' \). Assume by way of contradiction that \( op_i \) does not complete in \( r \). Therefore, \( op_i \) is either stuck in one of its waits or continuously iterates in a while loop. In each case, we show a contradiction. Denote by \( con \) the consensus associated with timestamp \( s h n \).ts. By definition of \( s h n \), no decision is made in \( con \) in \( r \).

- Operation \( op_i \) waits in line 16 (Algorithm 8) forever. Notice that \( s h n \).cng.rem contains all the process that were removed in \( r \), so, after time \( t' \), \( p_i \) does not wait for a reply from a removed process. By the strong completeness property of FD, \( p_i \) does not wait for faulty processes forever. A contradiction.

- Operation \( op_i \) waits in line 18 (Algorithm 8) forever. Notice that from time \( t' \) till \( p_i \) proposes in \( con \), \( pend_i=false \). Therefore, \( p_i \) proposes in \( con \) in line 22 (Algorithm 8), and waits in line 18 after the propose. By Observation 4, there is a majority \( M \) of \( s h n \).cng.em s.t. \( M \subseteq V(t).membership \cup P(t).join \setminus F(t) \). Therefore, by the termination of consensus, eventually a decision is made in \( con \). A contradiction to the definition of \( s h n \).

- Operation \( op_i \) remains in the while loop in line 17 (Algorithm 8) forever. Since it does not waits in line 18 (Algorithm 8) forever, \( op_i \) proposes infinitely many times, and since each propose is made in a different consensus and \( p_i \) can propose in a consensus beyond the first one only once a decision is made in the previous one, infinitely many decisions are made in \( r \). A contradiction to the definition of \( s h n \).
4.4. Conclusion

- Operation $op_i$ waits in line 27 (Algorithm 8) forever. Consider two cases. First, $sm_i \neq \hat{sm}$ when $p_i$ performs line 26 (Algorithm 8). In this case, $p_i$ continues at time $t'$, when it adopts $\hat{sm}$, because $sm_i.t_s > ts$ hold at time $t'$. In the second case ($sm_i = \hat{sm}$ when $p_i$ performs line 26), $p_i$ sends update message to all processes in $\hat{sm}.cng.mem$, and waits for a majority to reply. By Observation 4, there is a correct majority in $\hat{sm}.cng.mem$, and thus $p_i$ eventually receives the replies and continues. In both cases we have contradiction.

Therefore, $p_i$ completes in $r$. \hfill \Box

We conclude with the following theorem:

**Theorem 9.** Algorithms 7–9 implement wait-free atomic dynamic storage.

4.4 Conclusion

We proved that in an asynchronous API-based reconfigurable model allowing at least one failure, without restricting the number of reconfigurations, there is no way to emulate dynamic safe wait-free storage. We further showed how to circumvent this result using a dynamic eventually perfect failure detector: we presented an algorithm that uses such a failure detector in order to emulate a wait-free dynamic atomic MWMR register.

Our dynamic failure detector is (1) sufficient for this problem, and (2) can be implemented in a dynamic eventually synchronous [32] setting with no restriction on reconfiguration rate. An interesting question is whether a weaker such failure detector exists. Note that when the reconfiguration rate is bounded, dynamic storage is attainable without consensus, thus such a failure detector does not necessarily have to be strong enough for consensus.
Chapter 5

Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution

Our experience with building real-life distributed systems repeatedly surfaces reconfiguration as an important issue whose practice is less understood than desired. Providing clean and efficient foundations and tools for reconfiguration is therefore a crucial enabler for today’s distributed system management. We make a step towards providing such foundations in this work.

The goal of this work is to take a static fault-tolerant object like an atomic read/write register and turn it into a dynamic fault-tolerant one. A static object exposes an API (e.g., read/write) to its clients, and is emulated on top of a set of fault-prone servers (sometimes called base objects) via protocols like ABD [10]. We refer to the underlying set of fault-prone servers as a configuration. To convert a static object into a dynamic one, we first extend the object’s API to support reconfiguration. Such an API is essential for administrators, who should be able to remove old or faulty servers and add new ones without shutting down the service. One of the challenges in formalizing dynamic models is to define a precise fault condition, so that an administrator who requests to remove a server $s$ via a reconfiguration operation will know when she can switch $s$ off without risking losing the object’s state (e.g., the last written value to a read/write register).

To this end, we first define a clean dynamic failure model, in which an administrator can immediately switch a server $s$ off once a reconfiguration operation that removes $s$ completes. Then, we provide an abstraction for consensus-less reconfiguration in this model. To demonstrate the power of our Reconfiguration abstraction we use it to implement two dynamic atomic objects. First, we focus on the basic building block of a read/write register; thus, other (static) objects that can be emulated from read/write registers (e.g., atomic snapshots) can be made dynamic by replacing the underlying reg-
isters with dynamic ones. Second, we emulate a max-register [9], which on the one hand can be implemented asynchronously [10] (on top of fault-prone servers), and on the other hand cannot be emulated (for an unbounded number of clients) on top of a bounded set of read/write registers as we showed in Chapter 2. Thus, a standalone implementation of dynamic max-registers is required.

**Complexity.** We present an optimal-complexity implementation of our Reconfiguration abstraction in asynchronous environments, which in turn leads to the first optimal implementation of a dynamic read/write register in this model. More concretely, faced with \( n \) administrator reconfiguration requests, the number of configurations that the dynamic object is implemented over is \( n \); and the number of rounds (when the algorithm accesses underlying servers) per client operation is \( O(n) \). A comparison with previous solutions appears in Section 5.1.

**Dynamic fault model.** In Section 5.2 we provide a succinct failure condition capturing a versatile set of faults under which the dynamic object’s liveness is guaranteed. We define the dynamic fault model as an interplay between the object’s implementation and its environment: New configurations are introduced by clients, (which are part of the object’s environment). The object implementation then activates the requested configuration, at which point old configurations are expired. Between the time when a configuration is introduced and until it is expired, the environment can crash at most a minority of its servers. For example, when reconfiguring a register from configuration \( \{A, B, C\} \) into \( \{D, E, F\} \), initially a majority of \( \{A, B, C\} \) must be available to allow read/write operations to complete. Then, when reconfiguration is triggered, \( \{D, E, F\} \) is introduced, and subsequently, majorities of both configurations must be available, to allow state-transfer to occur. Finally, when the reconfiguration operation completes, leading to \( \{D, E, F\} \)’s activation, \( \{A, B, C\} \) is expired, and every server in it may be immediately shutdown.

**Reconfiguration abstraction.** Since a configuration is a finite set of servers, we can use ABD [10] to emulate in each configuration a set of (static) atomic read/write registers (as well as max-registers), which are available as long as the configuration is not expired. The Reconfiguration abstraction, in contrast, is not tied to a specific configuration, but rather abstracts away the coordination among clients that wish to change the underlying set of servers (configuration) emulating the dynamic object. Its specification, which is formally defined in Section 5.3, exposes two API methods, Propose and Check. Clients use Propose to request changes to the configuration, and Check to learn of changes proposed by other clients. Both return a configuration and a set of speculations. The returned configuration reflects all previous proposals and possibly some ongoing ones. The less obvious return value of Reconfiguration is the speculation set. This set is required since there is no guarantee that all clients see the same sequence of configurations (indeed,

---

1 A max-register for \( k \) clients requires at least \( k \) read/write registers.
Chapter 5. Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution

(a) Dynamic atomic read/write register on top of the Reconfiguration abstraction.

(b) Dynamic atomic max-register on top of the Reconfiguration abstraction.

Figure 5.1: The Reconfiguration abstraction usage. Solid (dashed) blocks depict dynamic (resp. static) objects.

Reconfiguration is weaker than consensus). Therefore, a dynamic object implementation that uses Reconfiguration needs to read from every configuration that Check returned to any other client, and transfer the most up-to-date value read in any of these to the new configuration returned from Check. To this end, Reconfiguration returns a speculation set that includes all configurations previously returned to all clients (and possibly additional proposed ones).

In Section 5.4, we implement (1) a dynamic atomic read/write register on top of the Reconfiguration abstraction and static atomic ranked registers [26] (one in every configuration), and (2) a dynamic atomic max-register on top of Reconfiguration and static atomic max-registers. See Figure 5.1 for illustrations. In Section 5.5 we give an optimal consensus-less algorithm for Reconfiguration, which together with the read/write register emulation of Section 5.4 yields an optimal dynamic read/write register algorithm.

In summary, this chapter makes three contributions: it defines a failure condition that allows an administrator to shutdown removed servers; it introduces the Reconfiguration abstraction, which captures the essence of reconfiguration; and it presents an asynchronous optimal-complexity solution for dynamic atomic registers. Section 5.6 concludes the chapter.
5.1 Related Work

Model. The problem of object reconfiguration has gained growing attention in recent years [41, 56, 8, 66, 50, 40, 72, 37, 49, 70, 11, 17]. However, dynamic failure models do not always make it clear when exactly an administrator can shutdown a removed server. Early works supporting dynamic objects [56, 41, 24] simply assume that a configuration is available as long as some client may try to access it. SmartMerge [50] uses a shared non-reconfigurable auxiliary object (lattice agreement) that is forever available to all clients, meaning that a majority of the servers emulating this auxiliary object can never be switched off. DynaStore [8] was the only previous work to define dynamic failure conditions based on a reconfiguration API, but these conditions are complicated, and restrict reconfiguration attempts as well as failures. Moreover, DynaStore does not separate clients from servers as we do here. Following [37, 50], we formulate the problem in shared memory, which makes it easier to reason about and clearer.

Other works [11, 17] assume a broadcast mechanism for announcing joins instead of an API for adding and removing processes, and bound the rate of changes of the underlying set of servers; the latter is necessary if one wants to ensure liveness for all operations (as [11] does) – as we show in Chapter 4 no asynchronous reconfigurable service can ensure liveness unless the reconfiguration rate is limited in some way. Like many earlier works [8, 37, 50], in this chapter we do not explicitly bound the reconfiguration rate, and hence ensure liveness only if the number of reconfigurations is finite.

Abstractions. All previous works have considered reconfiguration in some specific context – state machine replication [53, 19, 20], read/write register emulation [8, 50, 37, 41], or atomic snapshot [70]. To the best of our knowledge, this work is the first to specify general dynamic objects as extensions of their static counterparts and to provide a general abstraction for dynamic reconfiguration. We note that although [37] define a reconfigure primitive intended to capture the core reconfiguration problem, that primitive is not sufficiently strong for implementing an atomic register, (in particular, since it does not require real-time order), and indeed, they do not implement their atomic register on top of it.

Dynamic register complexity. In a recent non-refereed tutorial [72], we give a generic formulation that allows us to compare the complexity of different algorithms [41, 56, 50, 37, 8], as follows: Given that \( n \) is the number of proposed configuration changes and \( m \) is the total number of operations (read/write/reconfig) invoked on the atomic register, DynaStore [8] goes through \( O(\min(mn, 2^n)) \) configurations, and requires a constant number of operations in every configuration, so \( O(\min(mn, 2^n)) \) is also DynaStore’s operation complexity. Parsimonious SpSn [37] reduces the number of traversed configurations to \( O(n) \), but since they invoke a linear number of operations in every configuration, their total operation complexity is \( O(n^2) \).
Now notice that it is always possible to stagger reconfiguration proposals in a way that forces the system to go through \( \Omega(n) \) configurations. The asymptotically optimal \( O(n) \) operation complexity is straightforward to achieve in consensus-based solutions [41, 56, 24]. This complexity was also achieved by SmartMerge [50], but this was done using an auxiliary object that was assumed to be live indefinitely, i.e., was not reconfigurable in itself. Our algorithm is the first consensus-free and fully reconfigurable dynamic register algorithm with optimal complexity.

### 5.2 Dynamic Model

We consider a fault-prone shared memory model [48]: The system consists of an infinite set \( \Pi \) of clients (sometimes called processes), any number of which may fail by crashing, and an infinite set \( \Phi \) of servers (sometimes called base objects) supporting arbitrary atomic low-level objects. Clients access servers via low-level operations (e.g., read/write), which may take arbitrarily long to arrive and complete, hence the system is asynchronous.

We address in the chapter two atomic objects: a classical fault tolerant read/write register and a max-register [9]. Both registers provide clients with two API methods: Read and Write in case of read/write register, and MRead and MWrite in case of max-register. In a well-formed execution, a client invokes API methods one at a time, though calls by different clients may be interleaved in real time. For a well-formed execution, there exists a serialization of all client operations that preserves the operations’ real time order, such that (1) in case of read/write register a Read returns the value written in the latest Write preceding it, or \( \perp \) if there is no preceding Write; and (2) in case of max-register an MRead returns the highest value written by an MWrite that precedes it, or \( \perp \) if there is no preceding MWrite. (In case of max-registers, the values domain is ordered.)

**Configurations.** The universe of servers is infinite, but at any moment in time, a client chooses to interact with a subset of it. In our model, a configuration is a set of included and excluded servers, where configuration membership is the set of included and not excluded servers in the configuration. Formally:

\[
\begin{align*}
\text{Changes} & \triangleq \{+s \mid s \in \Phi\} \cup \{-s \mid s \in \Phi\} \\
\text{Configuration} & \triangleq \text{subset of Changes} \\
\text{C.membership} & \triangleq \{s \mid +s \in C \land -s \not\in C\}
\end{align*}
\]

For example \( C = \{+s_1, +s_2, -s_2, +s_3\} \) is a configuration representing the inclusion of servers \( s_1, s_2, \) and \( s_3, \) and the exclusion of \( s_2, \) and \( \text{C.membership} = \{s_1, s_3\} \). Tracking excluded servers in addition to the configuration’s membership is important in order to reconcile configurations suggested by different clients. The configuration size is the number of changes it includes— in this example, \( |C| = 4 \).
5.2. Dynamic Model

**Dynamic fault model.** A dynamic fault model is an interplay between the adversary’s power and the following events, which are invoked as part of client operations:

- **introduce(C):** indicates that C is going into use; and
- **activate(C):** indicates that the state transfer to C is complete.

By convention we say that the initial configuration $C_{init}$ is introduced and activated at time 0.

The above events govern the life-cycle of configurations. A configuration C becomes activated once an activate(C) event occurs. Note that not all introduced configurations are necessarily activated at some point. A configuration C becomes expired once activate(D) occurs s.t. C does not contain D. Intuitively, D reflects events (inclusions or exclusions) that are not reflected in C, and hence C has become “outdated”. Our algorithm will enforce a containment order among activated configurations, and will thus ensure that the latest activated one is not expired.

The following two conditions constrain the power of the adversary:

**Definition 20.** *(liveness conditions)*

**Availability:** The adversary can crash at most a minority of C.membership between the time when introduce(C) occurs and until C is expired.

**Weak Oracle:** When a client interacts with an expired configuration C, it either receives responses to calls from a majority of C.membership, or returns an exception notification $(\text{error}, D)$ for some activated D, where $C \not\supseteq D$.

Note that such an oracle (sometimes called directory service) is inherently required in order to allow slow clients to find non-expired configurations in an asynchronous system where old configurations may become unavailable [7, 70]. Our oracle definition is weak— in particular, the activated configuration it returns may itself be expired, and different clients may get different responses; it can be trivially implemented using a broadcast mechanism as assumed in some previous works [11, 17], and trivially holds if configurations must remain available as long as some client may access them, as in other previous works [41, 56, 24].

**Static versus dynamic objects.** A **static object** is one in which clients interact with a fixed configuration. In order to disambiguate a static object, scoped within a configuration C, from a dynamic one, we will label the methods of a static object with a “C.”. For every configuration C, as long as a majority of C.membership is alive, clients can use ABD [10] to emulate (static) atomic registers on top of the servers in C.membership. We denote:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C.x \leftarrow value$</td>
<td>A Write(value) operation to register x in configuration C</td>
</tr>
<tr>
<td>$C.x$</td>
<td>A Read of x</td>
</tr>
<tr>
<td>$C.collect(array)$</td>
<td>A bulk Read of all the registers in array</td>
</tr>
</tbody>
</table>
Since a complete array can be collected from servers using ABD in the same number of rounds as reading a single variable, we count a collect as a single operation for complexity purposes. Note that each register in the array is atomic in itself, but the collect is not atomic.

The methods of a dynamic object are not scoped with any configuration; it can start in some configuration and continue in a different one. A dynamic object’s API includes a ChangeConfig operation that allows clients to change the set of servers implementing the object. The implementation of ChangeConfig is object-specific, because it needs to transfer the state of the object across configurations, e.g., the last written value in case of an atomic register.

Clients pass to ChangeConfig a parameter $\text{Proposal} \subseteq \text{Changes}$ containing a proposed set of configuration changes. ChangeConfig returns a configuration $C$ s.t. (1) $C$ is activated, (2) $C \supseteq \text{Proposal}$, and (3) every configuration introduced or activated by ChangeConfig consists of $C_{\text{init}}$ plus a subset of changes proposed by clients before the operation returns.

The liveness guarantee of a dynamic object is that, assuming the number of ChangeConfig proposals is finite, every correct client’s operation eventually completes. Note that if the number of ChangeConfig proposals is infinite, it is impossible to ensure liveness for all operations as we show in Chapter 4.

Usage example. Consider an administrator (a privileged client) who wants to switch server $s$ off and invokes ChangeConfig($\{-s\}$). By liveness, ChangeConfig completes, and by properties (1) and (2), it returns an activated configuration $C \supseteq \{-s\}$. The activation of $C$ expires all configurations that do not contain $C$, and in particular, those that do not include $-s$. Hence, $s$ is not part of the membership of any unexpired configuration, and by the availability condition, the administrator can safely switch $s$ off immediately once ChangeConfig($\{-s\}$) returns.

5.3 Reconfiguration Abstraction

We introduce a generic reconfiguration abstraction, which can be used for implementing dynamic objects as we illustrate in the next section. A Reconfiguration abstraction has two operations:

$\text{Propose}(C, P)$ for a configuration $C$ and a proposed set of changes $P$; and

$\text{Check}(C)$ for a configuration $C$.

Propose is used to reconfigure the system, whereas Check is used in order to learn about other clients’ reconfiguration attempts. Propose and Check invoke the introduce and activate events. Both Check and Propose return a pair of values $\langle D, S \rangle$, where $D$ is a configuration and $S$ is a speculation set containing configurations; when $\langle D, S \rangle$ is returned we
say that \( D \) is nominated by the operation that returns it. Intuitively, a nominated configuration is one that has been introduced and is a candidate for activation. By convention, we say that \( C_{\text{init}} \) is nominated at time 0. We assume that the first argument passed to both operations is a nominated configuration.

The first property of Reconfiguration is validity, which (i) requires Propose\((C, P)\) to include \( P \) in the returned nominated configuration; and (ii) does not allow configurations to include spurious changes not proposed by any client. Formally:

\[
D_1 \text{ (Validity) (i) If Propose}\((C, P)\) returns \( \langle D, S \rangle \) for some \( S \), then \( D \supseteq P \), and (ii) for every configuration \( D \) that is introduced or nominated by an operation \( op \), for every \( e \in D \setminus C_{\text{init}} \), there is a Propose\((C', P')\) for some \( C' \) that is invoked before \( op \) returns s.t. \( e \in P' \).
\]

The second property ensures that nominated configuration sizes monotonically increase over time, which is essential for real-time order of operations invoked on objects that use this abstraction:

\[
D_2 \text{ (Real-time Order) A configuration } D \text{ nominated by operation } op \text{ is larger than or equal to every configuration nominated by an operation that strictly precedes } op.
\]

Since Reconfiguration is weaker than consensus, clients do not agree on a sequence of nominated configurations. Hence, in case some client \( c_1 \) proceeds to a configuration \( C' \), we want to ensure that if another client \( c_2 \) “skips” \( C' \), \( c_2 \) has \( C' \) in its speculation set, and can thus transfer any state that \( c_1 \) may have written there to the newer configuration \( c_2 \) nominates. This is captured by property \( S_1 \) below. Property \( S_1 \) stipulates that these configurations are also introduced, ensuring a live majority in these configurations in order to allow state transfer.

\[
S_1 \text{ (Speculation) If Check}\(C\) or Propose\((C, P)\) returns \( \langle D, S \rangle \), then every \( C' \in S \) is (i) introduced and (ii) \( S \) includes all nominated configurations \( C' \) s.t. \( |C| \leq |C'| \leq |D| \).
\]

As a practical matter, if any \( C'' \) between \( C \) and \( D \) has been activated, any \( C'' \) s.t. \( |C'| < |C''| \) may be omitted.

In addition, we have to define when configurations are activated. Note that an activation of a new configuration leads to expiration of old ones, and thus to possible loss of information stored in them. Therefore, a configuration \( D \) is not immediately activated when a Propose returns \( \langle D, S \rangle \) for some \( S \). Instead, a configuration \( C \) is activated if Check\(C\) does not report any newer configuration:

\[
A_1 \text{ (Activation) If Check}\(C\) returns \( \langle C, S \rangle \) for some \( S \), then \( C \) is activated.
\]

The liveness property of Reconfiguration is the same as in other dynamic objects \([8, 50, 37, 70]\), namely, if the number of Propose operations is finite, then every operation by a correct client completes.
5.4 Building Dynamic Objects Using Reconfiguration

We first present a dynamic atomic read/write register emulation using Reconfiguration, then explain the modifications needed for supporting a dynamic atomic max-register [9], and finally provide a formal proof.

5.4.1 Dynamic atomic read/write register

Besides the Reconfiguration abstraction, our dynamic register implementation uses a (static) ranked register [26] emulation in every configuration, as illustrated in Figure 5.1a. A ranked register stores a tuple, called version, that consists of a value $v$ and a monotonically increasing timestamp $ts$, and supports $RRRead()$ and $RRWrite(version)$ operations. The sequential specification of a ranked register is following: An $RRRead()$ returns the version with the highest $ts$ written by an $RRWrite$ that precedes it, or $\bot$ if there is no preceding $RRWrite$. Like all static objects in our model, if the configuration where the ranked register is emulated expires, the oracle returns an error.

The basic framework for implementing the Read, Write, and ChangeConfig operations is a loop: (i) Check, (ii) read (using RRread) the highest version from all speculated configurations returned by Check, (iii) write (with RRWrite) the highest version to the configuration nominated by Check, (iv) repeat. The loop terminates when Check does not nominate a new configuration. The specific action of each of the three operations is as follows. A Read simply returns the value of the highest version at the end of the loop. A Write increments the timestamp and writes it with a new value at the beginning of the loop. ChangeConfig proposes a configuration change via Propose instead of Check in the first iteration.
Algorithm 10 Dynamic atomic read/write register using Reconfiguration.

Client local variables:
1: configuration $C_{curr}$, initially $C_{init}$
2: $TS = \mathbb{N} \times \Pi$ with selectors $num$ and $id$
3: $version \in V \times TS$ with selectors $v$ and $ts$, initially $\langle v_0, (0, \text{client's id}) \rangle$
4: pickTS $\in \{\text{true}, \text{false}\}$, initially true.

Code for client $c_i \in \Pi$:

5: Read()
6: transferState($\text{Check}(C_{curr})$, $\bot$)
7: checkConfig()
8: return version.$v$

9: Write($v$)
10: transferState($\text{Check}(C_{curr})$, $v$)
11: checkConfig()
12: pickTS $\leftarrow$ true
13: return ok

14: ChangeConfig($P$)
15: transferState($\text{Propose}(C_{curr}, P)$, $\bot$)
16: checkConfig()
17: return $C_{curr}$

18: On $\langle \text{error}, D \rangle$ do
19: $C_{curr}$ $\leftarrow$ $D$
20: restart operation

21: procedure checkConfig()
22: $\langle D, S \rangle \leftarrow \text{Check}(C_{curr})$
23: while $D \neq C_{curr}$ do
24: transferState($\langle D, S \rangle$, $\bot$)
25: $\langle D, S \rangle \leftarrow \text{Check}(C_{curr})$

26: procedure transferState($\langle D, S \rangle$, value)
27: for each $C \in S$ do
28: $\text{tmp} \leftarrow C.\text{RRRead}()$
29: if $\text{tmp}.ts > version.ts$ then
30: version $\leftarrow \text{tmp}$
31: if value $\neq \bot$ $\lor$ pickTS $=$ true then
32: version $\leftarrow \langle \text{value}, \langle \text{version}.ts.num + 1, i \rangle \rangle$
33: pickTS $\leftarrow$ false
34: $D.\text{RRWrite}($version$)$
35: $C_{curr} \leftarrow D$

The pseudocode appears in Algorithm 10. The transferState method reads the register’s version from the entire speculation set $S$ and writes the latest version to the new configuration $D$. The checkConfig method repeatedly calls transferState until the configuration returned by Check stops changing. During the loop execution, an operation on an expired configuration may incur an exception, with a notification of the form $\langle \text{error}, D \rangle$ (see line 18). In this case, the loop is aborted and the operation starts over at configuration $D$. In case write is restarted after it has chosen a new timestamp, it skips the timestamp selection step.

We say that a configuration $C$ becomes stable when some version is written to $C$ in step (iii). We refer to the first version written to $C$ as the opening version of $C$. Consider a completed operation (Read, Write, or ChangeConfig) $op$ and let $C$ be the last configuration in which $op$ writes some version $v$, we say that $op$ commits $v$ in $C$ when it completes.

The correctness of the register emulation is based on the following key invariant:

**Invariant 2.** For every stable configuration $C$, the opening version of $C$ is higher than or equal to the highest version committed in any configuration $C'$ s.t. $|C'| < |C|$.

In other words, a larger stable configuration always holds a newer (or equal) version of
Chapter 5. Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution

the register’s value than that committed in a smaller activated one.

Algorithm 11 Dynamic atomic max-register using Reconfiguration.

Client local variables:
1: configuration \( C_{\text{curr}} \), initially \( C_{\text{init}} \)
2: \( value \in V \), initially \( v_0 \)

Code for client \( c_i \in \Pi \):

3: \( \text{MRead()} \)
4: \( \text{transferState}(\text{Check}(C_{\text{curr}}), \bot) \)
5: \( \text{checkConfig()} \)
6: \( \text{return value} \)

7: \( \text{MWrite}(v) \)
8: \( \text{transferState}(\text{Check}(C_{\text{curr}}), v) \)
9: \( \text{checkConfig()} \)
10: \( \text{return ok} \)

11: \( \text{ChangeConfig}(P) \)
12: \( \text{transferState}(\text{Propose}(C_{\text{curr}}, P), \bot) \)
13: \( \text{checkConfig()} \)
14: \( \text{return } C_{\text{curr}} \)

15: \( \text{On } \langle \text{error}, D \rangle \) do
16: \( C_{\text{curr}} \leftarrow D \)
17: \( \text{restart operation} \)

18: \( \text{procedure checkConfig()} \)
19: \( \langle D, S \rangle \leftarrow \text{Check}(C_{\text{curr}}) \)
20: \( \text{while } D! = C_{\text{curr}} \text{ do} \)
21: \( \text{transferState}(\langle D, S \rangle, \bot) \)
22: \( \langle D, S \rangle \leftarrow \text{Check}(C_{\text{curr}}) \)

23: \( \text{procedure transferState}(\langle D, S \rangle, v) \)
24: \( \text{if } v \neq \bot \text{ then} \)
25: \( \text{value } \leftarrow v \)
26: \( \text{for each } C \in S \text{ do} \)
27: \( \text{tmp } \leftarrow C.\text{MRead()} \)
28: \( \text{if } \text{tmp} > \text{value} \text{ then} \)
29: \( \text{value } \leftarrow \text{tmp} \)

30: \( D.\text{MWrite}(\text{value}) \)
31: \( C_{\text{curr}} \leftarrow D \)

Complexity. We measure complexity in terms of the number of accesses to low level objects, namely static atomic registers. Note that Read/Write/collection operations on static registers are emulated in a constant number of rounds using ABD. The complexity of the dynamic register’s operations is determined by (1) the complexity of the operations inside the Checks invoked during the loop (plus possibly one Propose); and (2) the sum of the sizes of all speculation sets returned by Propose/Check operations in this loop (where the register’s implementation performs Reads).

In a run with \( n \) ChangeConfig proposals, clearly, the best complexity we can hope for is \( O(n) \). In the next section we present our algorithm for Reconfiguration, which achieves the asymptotically optimal \( O(n) \) complexity.

5.4.2 Dynamic atomic max-register

The emulation of a max-register on top of Reconfiguration is similar to the read/write register emulation. It differs in how we keep and transfer the state, i.e., the register’s value. First, instead of a (static) ranked register in each configuration, we use a (static) max-register. Second, instead of timestamps, we use the actual written values, that is, a writer writes its value in step (iii) only if it is higher than all the values read in step
(ii) (Otherwise, it transfers the highest value it read). The pseudocode appears in Algorithm 11.

5.4.3 Read/write register correctness proof

We start with notations:

**Notation.** We say that a version \( v \) is higher than a version \( v' \) if \( v.ts > v'.ts \). An operation (Read, Write, or ChangeConfig) stores a version \( v \) in configuration \( C \) when it performs \( C.RRWrite(v) \) (line 34). A configuration \( C \) becomes stable when some operation stores a version in \( C \). We say that the opening version of a stable configuration \( C \) is the first version that is stored in \( C \). A configuration \( C \) holds a version \( v \) at time \( t \) if every \( C.RRRead() \) performed after time \( t \) returns a version that is higher than or equal to \( v \). Consider a completed operation (Read, Write, or ChangeConfig) \( op \) and let \( C \) be the last configuration in which \( op \) stores some version \( v \), we say that \( op \) commits \( v \) in \( C \) when it completes.

The following observation follows immediately from the specification of ranked register:

**Observation 12.** A stable configuration holds at time \( t \) the highest version that was stored in it before time \( t \).

The following observation follows immediately from the definitions of activated, stable and nominated configurations, and the code:

**Observation 13.** Every activated configuration is Stable and every stable configuration is nominated.

The following observations follow immediately from the code:

**Observation 14.** Every completed operation commits a version.

**Observation 15.** A Check and Propose are always called with a stable configuration.

We now ready to prove the correctness of our register, which rely on the following invariant:

**Invariant 2** (restated). For every stable configuration \( C \), the opening version of \( C \) is higher than or equal to the highest version committed in any configuration \( C' \) s.t. \(|C'| < |C|\).

**Proof.** Assume in a way of contradiction that there is a time \( t \) at which the invariant does not hold. Let \( C_{st} \) be the smallest stable configuration that violate the invariant at time \( t \), let \( v_{st} \) be the opening version of \( C_{st} \), and let \( op_{st} \) be the operation that stores \( v_{st} \) in \( C_{st} \). Now let \( t_1^{st} \) be the time when a Check or a Propose performed by \( op_{st} \) returns \((C_{st}, S)\) for some \( S \) for the first time. Denote this Check or Propose by \( CP_{st} \), and let \( t_2^{st} \) be the time when \( op_{st} \) invokes procedure \( transferState((C_{st}, S), \bot) \) (line 26). Note that \( t_1^{st} < t_2^{st} < t \).
By the contradiction assumption there is at least one version, committed in a smaller configuration than \( C \) until time \( t \), that is higher than \( v_{sl} \). Let \( op_c \) be an operation that commits version \( v_c < v_{sl} \) in configuration \( C_c \) until time \( t \) s.t. \( |C_c| < C_{sl} \). Now let \( t_1^c \) be the time when \( op_c \) stores \( v \) in \( C_c \) and let \( t_2^c > t_1^c \) be the time when \( op_c \) invokes \( \text{Check}(C_c) \) for the last time. Note that this \( \text{Check}(C_c) \) returns \( \langle C_c, S \rangle \) for some \( S \). Now consider two case:

- First, \( t_1^c < t_2^c \). Meaning that \( op_c \) invokes a \( \text{Check}(C) \) that nominates \( C \) after \( op_{sl} \) nominates \( C_{sl} \). A contradiction to property \( D_2 \) (Real time order) of the Reconfiguration abstraction.

- Second, \( t_1^c > t_2^c \). Therefore, \( t_2^c > t_1^c \), meaning that \( op_{sl} \) performs \( \text{transferState}(\langle C_{sl}, S \rangle, \bot) \) after \( op_c \) stores \( v \) in \( C_c \). Let \( C_{in} \) be the input configuration to \( CP_{sl} \). Now consider two case:
  
  - First, \( C \in S \). Since \( op_{sl} \) performs \( C_{sl}.RRRead() \) (in procedure \( \text{transferState} \)) after \( t_1^c \), and since \( \text{transferState} \) stores the highest version it reads, we get that \( v_{sl} \) is higher than or equal to \( v_c \). A contradiction.
  
  - Second, \( C \notin S \). By property \( S_1 \) (Speculation) of the Reconfiguration abstraction, this case is possible only if (1) \( |C_c| < |C_{in}| \leq |C_{sl}| \) and \( C_{in} \in S \) or (2) \( S \) includes an activated configuration \( C_a \) s.t. \( |C_c| < |C_a| \leq |C_{sl}| \). By Observations 13 and 15, \( C_a \) and \( C_{in} \) are stable. Therefore, in both cases, \( S \) includes a stable configuration \( C' \) s.t. \( |C_c| < |C'| \leq |C_{sl}| \). Now consider two case:
    
    1. First, \( |C'| = |C_{sl}| \), meaning that \( C_{sl} \) is stable at time \( t_1^c \). Therefore, some operation stores a version in \( C_{sl} \) before \( op_{sl} \). A contradiction to \( v_{sl} \) being the opening version of \( C_{sl} \).
    
    2. Second, \( |C'| < |C_{sl}| \). Since \( C_{sl} \) is the smallest stable configuration that violates the invariant at time \( t \), \( C_a \)’s opening version is higher than or equal to \( v_c \). By Observation 12, and since \( \text{transferState} \) stores the highest version it reads from configurations in \( S \), we get that \( v_{sl} \) is higher than or equal to \( v_c \). A contradiction.

\[\square\]

From Invariant 2 and Observation 12 we get the following corollary:

**Corollary 13.** At every time \( t \) a stable configuration \( C \) holds a version that is higher than or equal to the highest version committed in any configuration \( C' \) s.t. \( |C'| \leq |C| \) before time \( t \).

The following lemma follows from Corollary 13 and the specification of the reconfiguration abstraction:

**Lemma 41.** Consider an operation \( op_1 \) that commits a version \( v_1 \) in configuration \( C_1 \), and an operation \( op_2 \) that begins after \( op_1 \) returns. Let \( \text{Check}_2 \) be a Check performed by \( op_2 \), and let
\( \langle C_2, S_2 \rangle \) be the valued it returns. Then, \( S_2 \) includes a stable configuration that holds a version higher than or equal to \( v_1 \).

**Proof.** Note that \( op_1 \) nominates \( C_1 \), and since \( op_1 \) precedes \( op_2 \), \( op_1 \) performs a Check that nominates \( C_1 \) before \( op_2 \) invokes \( Check_2 \). Thus, by property \( D_2 \) (real time order) of the reconfiguration abstraction, \( |C_1| \leq |C_2| \). Moreover, by Observation 15 and property \( S_1 \) (speculation) of the reconfiguration abstraction, \( S_2 \) includes a stable configuration \( C_s \) s.t. \( |C_s| \geq |C_1| \). By Corollary 13, \( C_s \) holds a version with a timestamp \( ts_s \geq ts_1 \).

Notice that every completed Write operation picks exactly one timestamp (line 32), every picked timestamp is unique (ties are broken by clients ids), and there are no two different versions with the same timestamp. Thus, we say that Write operations are associated with the timestamp they pick. Note also that a Read operation returns the value in the version it commits. Therefore, we say that Read operations are associated with the timestamps of the versions they commits. The following Observation follows immediately from the code:

**Observation 16.** A completed Write operation that is associated with timestamp \( ts \) commits a version with timestamp \( ts' \geq ts \).

**Lemma 42.** Consider two completed Write operations \( w_1, w_2 \) that are associated with timestamps \( ts_1, ts_2 \), respectively. If \( w_1 \) precedes \( w_2 \), then \( ts_1 < ts_2 \).

**Proof.** Let \( ts_1^c \) be the timestamp of the version committed by \( w_1 \). By Observation 16, \( ts_1^c \geq ts_1 \). Let \( Check_2 \) be the last check \( w_2 \) performs before picking a timestamp, and let \( \langle C_2, S_2 \rangle \) be its return value. By Lemma 41, \( S_2 \) includes a stable configuration that holds a version with a timestamp \( ts_s \geq ts_1^c \geq ts_1 \). Therefore, by the code of \( transferState \), \( ts_2 > ts_s \geq ts_1^c \geq ts_1 \).

**Lemma 43.** Consider two completed Read operations \( rd_1, rd_2 \) that are associated with timestamps \( ts_1, ts_2 \), respectively. If \( rd_1 \) precedes \( rd_2 \), then \( ts_1 \leq ts_2 \).

**Proof.** Let \( Check_2 \) be the Check operation that \( rd_2 \) performs before storing a version with timestamp \( ts_2 \), and let \( \langle C_2, S_2 \rangle \) be the return value of \( check_2 \). By Lemma 41, \( S_2 \) includes a stable configuration that holds a version with a timestamp \( ts_s \geq ts_1 \). Therefore, by the code of \( transferState \), \( ts_2 \geq ts_s \geq ts_1 \).

**Lemma 44.** Consider a Write operation \( w_1 \) associated with timestamp \( ts_1 \), and a Read operation \( rd_2 \) associated with timestamp \( ts_2 \). If \( w_1 \) precedes \( rd_2 \), then \( ts_1 \leq ts_2 \).

**Proof.** Let \( ts_1^c \) be the timestamp of the version committed by \( w_1 \). By Observation 16, \( ts_1^c \geq ts_1 \). Let \( Check_2 \) be the Check operation that \( rd_2 \) performs before storing a version with timestamp \( ts_2 \), and let \( \langle C_2, S_2 \rangle \) be the return value of \( check_2 \). By Lemma 41, \( S_2 \)
includes a stable configuration that holds a version with a timestamp \( t_s \geq t_{s_1} \geq t_{s_2} \). Therefore, by the code of \( \text{transferState} \), \( t_{s_2} \geq t_s \geq t_{s_1} \).

**Lemma 45.** Consider a Read operation \( rd_1 \) associated with timestamp \( t_{s_1} \), and a Write operation \( w_2 \) associated with timestamp \( t_{s_2} \). If \( rd_1 \) precedes \( w_2 \), then \( t_{s_1} < t_{s_2} \).

**Proof.** Let \( \text{Cheak}_2 \) be the last check \( w_2 \) performs before picking a timestamp, and let \( (C_2, S_2) \) be its return value. By Lemma 41, \( S_2 \) includes a stable configuration that holds a version with a timestamp \( t_{s_1} \geq t_{s_1} \). Therefore, by the code of \( \text{transferState} \), \( t_{s_2} > t_s \geq t_{s_1} \).

**Definition 21** (lineraization). For every run \( r \) we define the sequential run \( \sigma_r \) as follows: All the Write operations in \( r \) are ordered in \( \sigma_r \) by the timestamp they are associated with, and every Read operation associated with a timestamp \( t_s \) in \( r \) is ordered in \( \sigma_r \) immediately after the Write that is associated with \( t_s \). Read operations that are associated with the same timestamps are ordered (among themselves) by the time they return.

**Theorem 10.** The algorithm emulates an atomic register.

**Proof.** We need to show that for every run \( r \), \( \sigma_r \) is a linearization of \( r \). The sequential specification is satisfied by construction, and the real time order follows from Lemmas 42, 43, 44, and 45.

## 5.5 The Reconfiguration Abstraction Implementation

In this section we present an optimal and modular Reconfiguration implementation. In Section 5.5.1 we introduce the Common Set (CoS) building block, which is used by the Reconfiguration abstraction in every configuration. In Section 5.5.2 we show how CoS is used for non-optimal Reconfiguration and in Section 5.5.3 we optimize the algorithm. Formal proofs for correctness and complexity are given in Sections 5.5.4 and 5.5.5, respectively.

### 5.5.1 CoS building block

The Common Set (CoS) building block is a static shared object, emulated in every configuration \( C \) over a set of (static) registers. Its API consists of a single operation, denoted \( C.CoS(P) \), where \( P \) is a set of arbitrary values. \( C.CoS \) returns an output set of sets satisfying the following:

**Definition 22** (Common Set in configuration \( C \)).

(CoS\(_1\)) Each set in the output is the union of some of the inputs and strictly contains \( C \);
5.5. The Reconfiguration Abstraction Implementation

Algorithm 12 Efficient CoS; algorithm of client $p_i$ in configuration $C$; optimization code shaded.

1: Local variables: $\triangledown$ flags accessible outside CoS
2: $\text{firstTime}$ set by reconfig and read by CoS
3: $\text{drop}$ set by CoS and read by reconfig

4: Shared variables (emulated in configuration $C$):
5: Boolean $\text{startingPoint}$, initially false $\triangledown$ Is $C$ a starting point for some client
6: Mapping from client to registers $\text{Warr}$ and $\text{Sarr}$, initially $\{}$

7: procedure $C.OS(P)$
8: $P \leftarrow \text{PreCompute}(P)$ $\triangledown$ optimization
9: if $P \supset C$ then
10: $\triangleright$ Something new to propose
11: $\text{C.Warr}[i] \leftarrow P$
12: return $\text{C.collect}($Warr$)$
13: else
14: return $\text{C.collect}($Warr$)$

16: procedure $\text{PreCompute}(P)$
17: if $\text{firstTime}$ then
18: $\text{C.startingPoint} \leftarrow \text{true}$
19: $\text{C.Sarr}[i] \leftarrow P$
20: $\text{drop} \leftarrow \text{false}$
21: if $\neg \text{C.startingPoint}$ then
22: return $P$
23: $\triangleright$ repeat collect until $P$ stops changing.
24: $\text{drop} \leftarrow \text{true}$
25: $\text{tmp} \leftarrow \bigcup \text{C.collect}($Sarr$)$
26: while $\text{tmp} \neq P$ do
27: $P \leftarrow \text{tmp}$
28: $\text{tmp} \leftarrow \bigcup \text{C.collect}($Sarr$)$
29: return $P$

(CoS$_2$) if a client’s input strictly contains $C$, then its output is not empty;

(CoS$_3$) there is a common non-empty set in all non-empty outputs; and

(CoS$_4$) every C.Cos invocation that strictly follows a C.Cos call that returns a non-empty output returns a non-empty output.

For example, consider three concurrent clients that input to C.Cos the sets $P_1$, $P_2$, and $P_3$, all of which contain $C$. A possible outcome is for their outputs to be $\{P_1\}$, $\{P_1, P_1 \cup P_2\}$, and $\{P_1, P_2, P_3\}$, respectively. The intuitive explanation behind using CoS is that it builds a common sequence of configurations inductively: The first configuration in the sequence is $C_{init}$, the next is the common configuration returned by $C_{init}.\text{CoS}$ (property CoS$_3$), and so on. Although this sequence is not known to the clients themselves, every client observes this sequence starting with some activated configuration. Every configuration in this sequence contains the previous one.

CoS can be implemented directly using consensus or atomic snapshot, as illustrated in [72]. In Algorithm 12, (without the PreCompute function, which is an optimization and will be discussed later), we give an implementation based on DynaStore’s weak snapshot [8]. In the pseudocode, we denote by $\bigcup S$ the union of all sets in a set of sets $S$. If the proposal $P$ strictly contains $C$, $p_i$ has something new to propose and it writes $P$ into its cell in the “weak” snapshot array $\text{Warr}$ (lines 9-10). (Note that $\text{Warr}$ is a static array emulated in the configuration where CoS is implemented). Either way, it collects $\text{Warr}$ (line 11). In case the collect is not empty, $p_i$ collects $\text{Warr}$ again and returns the set of collected proposals (lines 12-15). The second collect ensures that the intersection of non-
empty outputs includes the first written input, implying $\text{CoS}_3$; the remaining properties are satisfied by a single collect.

5.5.2 Simple Reconfiguration

Given $\text{CoS}$, we can solve Reconfiguration in a generic way as shown in Algorithm 13 (ignore the shaded areas for now). Both Check and Propose use the auxiliary procedure $\text{reconfig}$. Propose$(C, P)$ first sets a local variable $\text{proposal}$ to the union of $C$ and $P$, whereas Check$(C)$ initiates $\text{proposal}$ to be $C$. Both then execute the loop in line 40. Each iteration selects the smallest configuration in $\text{ToTrack}$; we say that the iteration tracks this configuration. The loop tracks all configurations returned by $\text{CoS}$, smallest to largest, starting with $C$. In each tracked configuration $C'$, the client introduces $C'$, invokes $C'.\text{CoS}(\text{proposal})$ and adds to $\text{proposal}$ the union of the configurations returned from $C'.\text{CoS}$. This repeats for every configuration $C'$ returned from $\text{CoS}$ until there are no more configurations to track. Recall that by the liveness condition, if some configuration $C'$ is expired and no longer supports $C'.\text{CoS}$, then the client gets in return to $C'.\text{CoS}$ an exception with some newer activated configuration $C_a$. In this case, $\text{reconfig}$ starts over from $C_a$. At the end, Propose and Check return $\text{proposal}$ and the set of all tracked configurations.

The common sequence starts with $C_{\text{init}}$, and is inductively defined as follows: If $C_k.\text{CoS}$ has a non-empty output, then $C_{k+1}$ is the smallest common configuration returned by all non-empty $C_k.\text{CoSs}$. By $\text{CoS}_3$, all non-empty return values have at least one configuration in common, and if there is more than one such configuration, then we pick the smallest, breaking ties using lexicographic order. By $\text{CoS}_1$, each configuration in the common sequence strictly contains the previous one.
Algorithm 13 Generic Reconfiguration algorithm; optimization code shaded.

29: Propose(C, P)
30: return reconfig(C, P)
31: Check(C)
32: ret ← reconfig(C, \{\})
33: if ret = ⟨C, ∗⟩ then activate(C)
34: return ret
35: procedure reconfig(C, P)
36: proposal ← P ∪ C
37: ToTrack ← \{C\}
38: speculation ← \{\}
39: firstTime ← true
40: while ToTrack ≠ \{\} do
41: C′ ← argmin C′′ ∈ ToTrack | C′′ | ⊳ smallest configuration
42: introduce(C′)
43: speculation ← speculation ∪ \{C′\}
44: ret ← C′.CoS(proposal)
45: if ret = ("error", C0) then  ⊳ C′ is expired - restart from C0
46: return reconfig(C0, proposal)
47: ToTrack ← (ToTrack ∪ ret) \ {C′}
48: firstTime ← false
49: if drop = true then  ⊳ drop old configurations in ToTrack
50: ToTrack ← ret
51: proposal ← proposal ∪ ToTrack
52: C curr ← proposal
53: return (proposal, speculation)

Correctness. The validity property (D1) immediately follows from CoS property CoS1 and the observation that proposal is set to include P at beginning of reconfig and never decreases.

To provide intuition for the remaining properties, we discuss the case in which all operations start in C init and no exceptions occur; the proof for the general case appears in Section 5.5.4 Observe that since proposal always contains ∪ ToTrack and configurations are traversed from smallest to largest, we get from property CoS2 that C.CoS returns an empty set only if C includes ToTrack, i.e., C is the last traversed configuration. The key correctness argument is that all nominated configurations belong to the common sequence, and are thus related by containment:

Lemma 46. For every reconfig that returns (D, S), D belongs to the common sequence.

Proof - sketch for the special case (starting in C init, no exceptions). Assume by way of contradiction that D j is returned by reconfig operation rec j but does not belong to the common sequence. Note that C init is in the common sequence and is tracked by rec j. Let C j be the last configuration tracked by rec j that belongs to the common sequence. By assumption, C j ≠ D j, and thus, rec j gets a non-empty output from C j.CoS (it gets an output since we assume that there are no exceptions). But, this output includes some configuration in the common sequence, so rec j tracks a configuration in the common sequence after C j. A contradiction.
Liveness follows since (i) every call to CoS returns, either successfully or with an exception; and (ii) tracked configurations are monotonically increasing, and, provided that the number of reconfigurations is finite, they are bounded.

5.5.3 Optimal Reconfiguration

The key to the efficiency of our new algorithm is in its thrifty CoS implementation, and the signals it conveys to the reconfiguration algorithm, which minimize the number of tracked configurations. To this end, the efficient solution for CoS shares (local) state variables $firstTime$ and $drop$ with the Reconfiguration implementation.

To explain the intuition behind our algorithm, let us first consider a scenario in which all clients invoke register operations (Read, Write, or ChangeConfig) in the same starting configuration $C_0$ (e.g., $C_0$ may be $C_{init}$), and no exceptions occur. If $n$ of the clients invoke Propose, then there are $n$ sets $P_1, \ldots, P_n$ proposed by $reconfig(C, P_i)$ operations. The unoptimized (weak snapshot-based) CoS may return up to $2^n$ different subsets in CoS responses (assuming many clients invoke Read/Write operations), inducing high complexity.

Our algorithm reduces this complexity by running a pre-computation phase in Pre-Compute, which imposes a containment order on all configurations passed to, and hence returned from, CoS. This is done by running a variant of (strong) atomic snapshot $\lceil \rceil$ on all client proposals in configuration $C_0$. Specifically, each process writes its own proposal $P$ (line 19) to the “strong” array $Sarr$, and then (lines 24-27) repeatedly collects the union of all $Sarr$ cells into $P$, until $P$ stops changing. Like an atomic snapshot, this ensures that all results of PreCompute are related by containment. Note, however, that unlike an atomic snapshot, the complexity of this pre-computation is linear in the number of different proposals written, rather than in the number of participating processes; if collect encounters a newly written value that does not change the union of written values, Pre-Compute returns. In case all operations start in $C_0$, there are no new proposals in other configurations, and so the containment order is preserved throughout the computation. This ensures that the number of different configurations tracked by all clients is at most $n$.

Next, we account for the case that clients invoke (or restart due to exceptions) their operations in different starting configurations. We have to identify configurations where some client starts, and run PreCompute in them too. To this end, we have clients signal (by raising the startingPoint flag) if a configuration is their starting point. Every client that later runs $C.CoS$ sees this flag true, and executes the pre-computation. If a client $p_i$ sees the flag false in $C.CoS$, $p_i$ does not run the pre-computation. Nevertheless, since $p_i$ checks the flag after writing its value to $Sarr$, $p_i$’s proposal is already in the array before new clients that start in this configuration perform their collects, and so $p_i$’s proposal is contained in theirs. Thus, at this new starting point, all clients obtain proposals that are related by containment among themselves.

The tricky part is that old proposals that were included in $ToTrack$ before the new
starting point are not necessarily ordered relative to ensuing proposals, as in the following scenario:

- Clients $p_1$ and $p_2$ start in $C_0$ and propose $C_0 \cup \{+a\}$ and $C_0 \cup \{+b\}$, respectively; $p_1$ gets $\{C_1\}$, where $C_1 = C_0 \cup \{+a\}$, from $C_0$. $CoS$ and $p_2$ gets $\{C_1, C_2\}$, where $C_2 = C_0 \cup \{+a, +b\}$.
- Client $p_1$ tracks $C_1$, gets an empty set from $C_1$. $CoS$, and activates it. Client $p_3$ starts in $C_1$ (which is now activated), proposes $C_3 = C_1 \cup \{+c\}$ in $C_1$. $CoS$, and gets $\{C_3\}$.
- Later, $p_2$ tracks $C_1$, and gets $C_3$ in $C_1$. $CoS$’s output. At this point $p_2$’s $ToTrack$ contains $C_2$ and $C_3$, neither of which contains the other.

To achieve linear complexity, we have clients drop all configurations previously returned from $CoS$ at all the starting points they encounter. One subtle point is ensuring safety in the presence of such drops, and our proof of the general case of Lemma 46 addresses this issue.

Intuitively, since the purpose of tracking all configurations is to ensure that clients traverse the common sequence, once we know $C$ is in the common sequence, there is no need to continue to track any configuration older than $C$. So, the drop is safe.

A second subtle point is preserving linear complexity despite executing PreCompute in multiple starting points. But since (i) the worst-case complexity of a single pre-computation is linear in the number of different proposals written to it, (ii) each $CoS$ begins with a proposal that reflects all those seen in previous $CoS$s, and (iii) there are $n$ new proposals overall, the combined complexity of all pre-computations is $O(n)$.

Finally, we provide intuition for the complexity of the high-level dynamic atomic register given in Section 5.4. The full proof, which wraps this intuition into a technical induction, appears in the next sections. Recall that the register emulation performs a loop in which it repeatedly calls Check($C$), where $C$ is the configuration returned from the previous Check/Propose, until some Check($C'$) returns $\langle C', S \rangle$ for some $C'$ and $S$. The loop performs a constant number of operations in every configuration returned in a speculated set $S$ from Check. Therefore, we want the Checks in this loop to return the optimal number of configurations, and have optimal complexity themselves.

Since all the configurations introduced (and returned in speculation sets) by our algorithm are related by containment, we immediately conclude that the number of configurations returned in speculated sets $S$ of all Checks together is bounded by $n$. Now we show that the complexity of all Checks combined is $O(n)$. First observe that all Checks combined invoke at most $n$ $CoS$s. Second, each $CoS$ writes at most three times to shared registers (lines 10, 18, and 19), reads once (in line 21), and performs each of the collects in lines 11, 15, and 24 at most once. Now observe that $CoS$ performs the collect in line 27 only if the previous collect (in line 24 or 27) contained a proposal $P_1 \not\subseteq P$, which means that none of the $CoS$s collected $P_1$ before. Since there are at most $n$ proposals, all $CoS$s together perform the collect in line 27 at most $n$ times. All in all, we get that the complexity of all Checks is $O(n)$.
5.5.4 Reconfiguration Correctness Proof

The proof makes use of the following simple observation:

Observation 17. The output of PreCompute contains its input.

Lemma 47. Algorithm 12 implements CoS.

Proof. We show the four properties of Definition 22:

\textbf{CoS}_1. Each CoS’s output is a set of proposals all of which were returned from PreCompute and, by line 9, strictly contain C. In PreCompute, P is always a subset of the union of inputs to CoS.

\textbf{CoS}_2. Consider a configuration C and a client \( p_i \) that invokes \( C . \text{CoS}(P) \) s.t. \( P \supset C \). By Observation 17, \( P \supset C \) also in line 9. Therefore, \( p_i \) writes P into \( \text{Warr}[i] \) (line 10), and since no other client writes into the same cell, \( p_i \) collects a non-empty set in its collects (lines 11 and 15). Thus, \( p_i \) returns a non-empty set.

\textbf{CoS}_3. If there are no non-empty outputs, then we are done. Otherwise, there is at least one client that writes its P to \( \text{Warr} \). Let \( p_i \) be the first such client (because \( \text{Warr} \) consists of atomic registers and atomicity is composable, the first is well-defined), and denote its P as \( P_i \). We next show that every returned non-empty set contains \( P_i \).

Consider a client \( p_j \) that returns a non-empty set. Then \( p_j \) collects \( \text{Warr} \) twice, and the first collect is not empty. Therefore, \( p_j \) completes its first collect after \( p_i \) writes \( P_i \) into \( \text{Warr}[i] \), and thus, it is guaranteed that \( p_j \) reads \( P_i \) from \( \text{Warr}[i] \) during its second collect, and returns a set that contains \( P_i \).

\textbf{CoS}_4. Consider two complete C.CoS calls \( op_i \) and \( op_j \) s.t. \( op_j \) strictly follows \( op_i \), and \( op_j \) returns a non-empty set. There is a non-empty cell in \( \text{Warr} \) before \( op_j \) completes, and since nothing is erased from \( \text{Warr} \), \( op_j \)’s collects are not empty.

We continue with the following notations:

Notation. We start with some notation that we use throughout the proof. The common sequence of configurations \( C_{\text{init}}, C_1, C_2, ..., \) which only an outside observer can view, is inductively defined as follows: It starts with \( C_{\text{init}} \), and if \( C_k . \text{CoS} \) has a non-empty output, then \( C_{k+1} \) is the smallest common configuration returned in all non-empty \( C_k . \text{CoS} \) outputs (by CoS property \( \text{CoS}_3 \), the intersection is not empty), with ties broken in lexicographic order.

We say that a set is monotonic if all its elements are related by containment. A sequence is monotonic if every element \( C^k \neq C^0 \) contains \( C^{k-1} \). By CoS property \( \text{CoS}_1 \), the
common sequence is monotonic. For a \textit{reconfig}(C, P) operation \textit{rec}_j, we say a configuration is \textit{tracked} by \textit{rec}_j if it is selected as \(C'\) in line 41. We define \(\text{tracked}(j) = C_0^j, C_1^j, \ldots, C_m^j\) to be the sequence of configurations tracked by \textit{rec}_j in the last recursive call of \textit{reconfig} in the order they are tracked. Note that \(\text{tracked}(j)\) does not include configurations tracked before an exception is received. We further denote \textit{rec}_j’s return value by \(\langle D_j, S_j \rangle\).

The following observations follow immediately from the code:

**Observation 18.** Consider a \textit{reconfig} operation \textit{rec}_j, then \(D_j = C_m^j \supseteq C_0^j \cup \cdots \cup C_m^j\).

**Observation 19.** If some \textit{reconfig} reads \textit{drop}=true in \(C.\text{CoS}\) at time \(t\), then there is a \textit{reconfig} \textit{rec}_j s.t. \(C_0^j = C\) that starts before time \(t\).

The next claim stipulates that \textit{reconfig} returns once \textit{CoS} returns it an empty set.

**Claim 5.** Consider a \textit{reconfig} operation \textit{rec}_j and a configuration \(C' \in \text{tracked}(j)\). If \(C'.\text{CoS}\) called during \textit{rec}_j returns an empty set, then \(C' = D_j\).

**Proof.** By \textit{CoS} property \(\text{CoS}_2\), since \(C'.\text{CoS}\) returns an empty set, when \(C'.\text{CoS}\) is called (line 44), \(\text{proposal} \not\supset C'\). Moreover, whenever \textit{CoS} is called during a \textit{reconfig} operation, \(\text{proposal} \supseteq \bigcup \text{ToTrack}\). Together, we get that \(C'\) is not strictly contained in \(\bigcup \text{ToTrack}\). Now notice that \(C'\) is selected in line 41 as \(\text{argmin}_{C \in \text{ToTrack}} |C|\). Therefore, we get that \(\text{ToTrack} = \{C'\}\) when \(C'.\text{CoS}\) is called (line 44). By the assumption, \(C'.\text{CoS}\) returns \({}\). Hence, in line \text{line:ToTrackUpdate}1 50/47, \text{ToTrack} becomes \({}\), and thus the \textit{reconfig} exits the while loop, and \(C' = D_j\).

The following observation follows from the usage of \textit{reconfig} and the oracle behavior:

**Observation 20.** For every \textit{reconfig} operation \textit{rec}_j, \(C_0^j\) is a nominated configuration that is returned by some \textit{reconfig} before \textit{rec}_j is invoked.

The next claim shows that our algorithm drops old configurations only upon tracking a configuration in the common sequence.

**Claim 6.** Assume that for every \textit{reconfig} operation \textit{rec}_j that returns before some time \(t\), \(D_j\) belongs to the common sequence. Now consider a configuration \(C\) that belongs to the common sequence, and a \textit{reconfig} operation \textit{rec}_j that tracks \(C\) and returns at time \(t\). If \textit{rec}_j gets a non-empty output from \(C.\text{CoS}\), then \textit{rec}_j tracks another configuration belonging to the common sequence after \(C\).

**Proof.** First observe that \textit{rec}_j gets a configuration \(C'\) that belongs to the common sequence from \(C.\text{CoS}\), and \textit{rec}_j’s \text{ToTrack} contains \(C'\) at the end of the corresponding while loop. If \textit{rec}_j tracks \(C'\), we are done. Otherwise, \textit{rec}_j drops \(C'\) after calling some \(C''.\text{CoS}\) (while tracking \(C''\)). By Observation 19, there is a \textit{reconfig} \textit{rec}_j s.t. \(C_0^j = C''\) that starts before time \(t\). By our assumption and by Observation 20, \(C''\) belongs to the common sequence, and we are done.
We now show that every nominated configuration belongs to the common sequence.

**Theorem 46** (restated). For every reconfig that returns \((D, S)\), \(D\) belongs to the common sequence.

**Proof.** We prove by induction on time \(t \geq 0\) that for every reconfig operation \(rec_j\) that returns at time \(t\), \(D_j\) belongs to the common sequence.

**Base:** Since no operation returns at time 0 or earlier, the lemma holds for \(t = 0\).

**Step:** We now assume that the lemma holds for some \(t \geq 0\), and prove for \(t + 1\). Let \(rec_j\) be a reconfig operation that returns at time \(t\). By Observation 20, \(C^0_j\) was returned by some reconfig before time \(t\). Therefore, by the induction assumption \(C^0_j\) belongs to the common sequence. Now assume in a way of contradiction that \(D_j\) does not belong to the common sequence. Let \(C\) be the last configuration tracked by \(rec_j\) that belongs to the common sequence (there is such a configuration since \(C^0_j\) belongs to the common sequence). By the contradiction assumption, \(C \neq D_j\), and thus by Claim 5, \(rec_j\) gets a non-empty output from \(C\).CoS. Therefore, by Claim 6, \(rec_j\) tracks a configuration belonging to the common sequence after \(C\). A get a contradiction.

The following is an immediate conclusion from Lemma 46 and the monotonicity of the common sequence.

**Corollary 14.** The set of nominated configurations is monotonic.

**Claim 7.** Consider a reconfig\((C, P)\) operation \(rec_j\) that returns \(D_j\). Then \(rec_j\) tracks every configuration in the common sequence between \(C\) and \(D_j\).

**Proof.** Assume by way of contradiction that there is a configuration \(C'\) in the common sequence between \(C\) and \(D_j\) that \(rec_j\) does not track. Let \(C^0_j\) be the last configuration before \(C'\) in the common sequence that is tracked by \(rec_j\). (There must be such a configuration because \(C = C^0_j\) is in the common sequence.) By Claim 5, \(rec_j\) gets a non-empty output from \(C^0_j\).CoS, and by the common sequence definition, this output contains the next configuration \(C_{\text{next}}\) in the common sequence. Now recall that \(rec_j\) tracks configurations from smallest to largest, and it does not track \(C_{\text{next}}\). Therefore, \(rec_j\) drops \(C_{\text{next}}\) after calling CoS in some configuration \(\tilde{C}\) not in the common sequence. By Lemma 46, \(\tilde{C}\) is not nominated, and thus not activated. Therefore, by the oracle definition no reconfig starts in \(\tilde{C}\), and thus the drop flag in \(\tilde{C}\).CoS is always false. Hence, \(rec_j\) does not drop configurations after calling \(\tilde{C}\).CoS. A contradiction.

**Theorem 11.** Algorithms 13 and 12 implement the Reconfiguration abstraction.

**Proof.** We now show that all Reconfiguration properties are satisfied:
(i) Consider a Propose\((C, P)\) operation that calls \(\text{reconfig}(C, P)\) and returns \(\langle D, S \rangle\). We have to show that \(D \supseteq P\). Now observe that \text{proposal} is set to contain \(P\) at the beginning of the \text{reconfig}, and never decreases, including recursive calls to \text{reconfig}. The property follows from line 53.

(ii) Consider a Check or a Propose operation \(op\) that introduces, activates, or return configuration \(C'\), and consider \(e \in C' \setminus C \cup P\). Observe that \(op\) gets \(e\) by some CoS output. The property follows by inductively using CoS\(_1\).

Consider an operation (Propose or Check) \(op_j\) that strictly precedes another operation \(op_i\). Let \(rec_j\) be the \text{reconfig} operation that is called during \(op_j\), and let \(rec_i\) be the \text{reconfig} operation that is called during \(op_i\). Assume by contradiction that \(|D_i| < |D_j|\). Thus, \(D_j \supset D_i\), and \(D_j\) precedes \(D_i\) in the common sequence. This means that \(D_i,\text{CoS}\) returned a non-empty output to some \text{reconfig} operation before \(D_j\) was added to the common sequence, and so before \(rec_i\) returned \(D_i\). Since \(D_j\) strictly precedes \(rec_i\), we get that \(D_i,\text{CoS}\) returned a non-empty output before \(rec_i\) invoked \(D_i,\text{CoS}\), and so by CoS\(_4\), \(D_i,\text{CoS}\) returns a non-empty output also to \(rec_i\), a contradiction to the assumption that \(rec_i\) returns \(D_i\).

Consider a Check\((C)\) or Propose\((C, P)\) operation \(op\) that returns \(\langle D, S \rangle\). Let \(rec_i = \text{reconfig}(C', P')\) be the last \text{reconfig} operation that is called during \(op\). By the oracle definition, by Lemma 46, and since every activate configuration is also nominated, \(\text{reconfig}(C, P)\) recursively calls \(\text{reconfig}(C_a, *)\) only if \(C_a\) is activated and nominated, and \(|C_a| > |C|\). Thus, if \(C' \neq C\), then \(C'\) is activated and belongs to the common sequence. By Lemma 46, all the nominated configurations are in the common sequence. Therefore, by Claim 7 and by the observation that the speculation set of \(rec_i\) is tracked\((i)\), \(S\) includes all nominated configurations \(C''\) s.t. \(|C'| \leq |C''| \leq |D|\).

5.5.5 Reconfiguration Complexity Proof

We use the following additional notations:

**Notation.** For every introduced configuration \(C\):

1. We define \(\text{OldProps}(C)\) to be the proposals suggested in \(C,\text{CoS}\) by \text{reconfig} operations starting their traversals before \(C\). That is,

\[
\text{OldProps}(C) \triangleq \{ P \mid \exists \text{reconfig} \text{that calls } C,\text{CoS}(P) \text{ while its flag firstTime} = \text{false} \}
\]

2. Since by Corollary 14, the set of nominated configurations is monotonic, there is at most one nominated configuration of a given size. Thus, we can define \(\text{Pred}(C)\)
to be the biggest nominated configuration $C'$ s.t. $|C'| < |C|$ for $C \neq C_{\text{init}}$. For completeness, we define $\text{Pred}(C_{\text{init}}) \triangleq C_{\text{init}}$.

For every nominated configuration $C'$:

1. All introduced configurations that have $C'$ as their Pred are its successors:

\[
\text{Successors}(C') \triangleq \{C'\} \cup \{C \text{ is introduced } | \text{ } C' = \text{Pred}(C)\}, \quad \text{and} \\
\text{successors}^i(C) \triangleq \{C' \in \text{successors}(C) | |C'| \leq i\}.
\]

2. All configurations that are in $\text{ToTrack}$ of reconfig operations after tracking $C'$ are potential successors of $C'$:

\[
\text{PotentialSuccessors}(C') \triangleq \{C \mid \text{some reconfig calls } C'.CoS \text{ in line } 44 \text{ and } C \in \text{ToTrack \ in line } 51 \text{ in the same iteration}\}
\]

An illustration of $\text{Successors}$ and $\text{PotentialSuccessors}$ appears in Figure 5.2.

We denote $\text{introducedSet}^i \triangleq \{C \text{ is introduced } | |C| \leq i\}$, and $\text{nominatedSet}^i \triangleq \{C \text{ is nominated } | |C| \leq i\}$.

![Figure 5.2: Example run of client $p_i$ of our algorithm. The dashed configuration is in $p_i$'s $\text{ToTrack}$ after calling $N_1.\text{CoS}$, and thus it is in $\text{potentialSuccessors}(N_1)$. But it is dropped after $p_i$ calls $N_2.\text{CoS}$, and thus it is never introduced, and so it is not in $\text{successors}(N_2)$.](image)

We now state some observations that follow immediately from the code.

**Observation 21.** Whenever $\text{CoS}$ is called during a reconfig operation, proposal $\supseteq \bigcup \text{ToTrack}$.

**Observation 22.** Consider a reconfig operation $\text{rec}$ that calls $C.\text{CoS}(P)$ s.t. $P \supseteq C$ and returns $\text{ret}$. Then $\exists C' \in \text{ret} \text{ s.t. } P \subseteq C'$. 
Observation 23. The return value of every PreCompute that reads startPoint=false in C.CoS is in OldProps(C).

Observation 24. If a configuration C' is returned by C.CoS, then there is a C.CoS invocation in which PreCompute returns C'.

Corollary 15. If all reconfig operations that call C.CoS read startPoint=false, then all C.CoS's return values in OldProps(C).

Observation 25. Let P_1 and P_2 be two proposals returned by PreCompute executions during C.CoS pc_1 and pc_2, respectively. If pc_1 and pc_2 execute lines 24 to 27 (repeat collecting Sarr until P stops changing), then P_1 and P_2 are related by containment.

Claim 8. Consider configuration C. If OldProps(C) is monotonic, then all configurations returned from C.CoS invocations are related by containment.

Proof. By Observation 24, we need to show that all proposals returned from PreCompute invoked during C.CoS are related by containment. Let P_1 and P_2 be two proposals returned by PreCompute executions during C.CoS pc_1 and pc_2, respectively. We show that P_1 and P_2 are related by containment. Consider three cases:

1. First, pc_1 and pc_2 read startPoint=false in line 21 executions during C.CoS. By Observation 23, P_1, P_2 ∈ OldProps(C). Since by the assumption OldProps(C) is monotonic, we are done.

2. Second, pc_1 reads startPoint=false, and pc_2 reads startPoint=true. Note that since pc_1 reads startPoint=false, it is called with P_1. Now since startPoint never changes from true to false, pc_1 reads startPoint before pc_2. Thus, pc_1 writes P_1 into Sarr in line 19 before pc_2 reads startPoint=true. Therefore, pc_2 sees P_1 in all the collects in lines 24 to 27, and thus P_2 ⊇ P_1.

3. Third, pc_1 and pc_2 read startPoint=true. Therefore, both execute lines 24 to 27, and thus by Observation 25, P_1 and P_2 are related by containment.

Claim 9. If a configuration C is nominated, then C is introduced.

Proof. By Lemma 46, C belongs to the common sequence, and so by the common sequence definition C is introduced.

Claim 10. Consider a reconfig operation rec_j that introduces configuration C' with firstTime=true and later introduces C. Then, C' ⊂ C.

Proof. Note that when C' is introduced, ToTrack = \{C'\}. Recall that if an error is returned, then rec_j is aborted, and no later configurations are introduced by rec_j. Thus, no error is returned when rec_j introduces C', and by CoS property CoS_{C_1}, at the end of the iteration
all the configurations in ToTrack are strictly contain C’. Therefore, the claim follows by inductively repeating this argument.

**Corollary 16.** Consider an introduced configuration C. Then, C ≥ C_init.

**Proof.** Let rec_j be a reconfig(C’, P’) that introduces C. Observe that rec_j introduces C’ with firstTime=true. Thus, by Claim 10, C ≥ C’. By Lemma 46, C’ belongs to the common sequence, and by monotonicity of the common sequence, C’ ≥ C_init. Therefore, C ≥ C_init.

**Claim 11.** Consider a reconfig operation rec_j that introduces configuration C in iteration iter. If firstTime=false at the beginning of iter, then there is at least one nominated configuration that is smaller than C, and rec_j tracks Pred(C) before iter.

**Proof.** Let C’ be the configuration that rec_j is called with. Since C’ and Pred(C) are nominated, by Corollary 14, C’ and Pred(C) belong to the common sequence. Now observe that rec_j introduces C’ with firstTime=true before it introduces C. Thus, by Claim 10, |C’| < |C|. If C’ = Pred(C), we are done, Otherwise, |C’| < |Pred(C)| < |C|. The Claim follows from Claim 7, and the observation that rec_j tracks configurations from the smallest to the biggest.

**Corollary 17.** Consider an introduced configuration C ≠ C_init. Then there is at least one nominated configuration that is smaller than |C| and a reconfig operation that tracks Pred(C) and introduces C.

**Proof.** Let rec_j be a reconfig operation that introduces C in an iteration iter, and consider two cases:

1. First, flag firstTime=false at the beginning of iter during rec_j. The Corollary follows by Claim 11.

2. Otherwise, C is the parameter rec_j is called with, and thus C nominated. Let rec_i be the first reconfig that tracks C, and let t be that time. Therefore, no reconfig returns C before time t. hence, rec_i is not called with C, and thus firstTime=false when rec_i tracks C. The Corollary follows by Claim 11.

**Claim 12.** Consider a nominated configuration C, and an introduced configuration C_{i+1} ∈ successors(C) of size i + 1 > |C|. Assume that for every C’ ∈ successors(C), PotentialSuccessors(C’) ⊆ PotentialSuccessors(C), and PotentialSuccessors(C) is monotonic. Then:

(a) C_{i+1} ∈ PotentialSuccessors(C)

(b) OldProps(C_{i+1}) ⊆ PotentialSuccessors(C)
Proof. (a) Since $C^{i+1} \neq C_{\text{init}}$, by Corollary 17, there is a \texttt{reconfig} operation $\texttt{rec}_i$ that tracks $C = \text{Pred}(C^{i+1})$ and introduces $C^{i+1}$. Now let $C^{n+1}$ be the biggest configuration in $\text{successors}^i(C)$ that is tracked by $\texttt{rec}_i$. By the assumptions $\text{PotentialSuccessors}(C^{n+1}) \subseteq \text{PotentialSuccessors}(C)$, and thus monotonic. Therefore, there is at most one configuration in $\text{PotentialSuccessors}(C^{n+1})$ whose size is $i + 1$. And since a \texttt{reconfig} operation tracks configurations by the order of their sizes, there is at least one configuration in $\text{PotentialSuccessors}(C^{n+1})$ whose size is $i + 1$. Therefore, there is exactly one configuration $C^{n+1}$ in $\text{PotentialSuccessors}(C^{n+1})$ of size is $i + 1$, and $\texttt{rec}_i$ tracks $C^{n+1}$ immediately after $C^{i+2}$. By CoS property $\text{CoS}_1$, $C^{n+2} = C^{i+2}$, and we are done.

(b) Consider some $P \in \text{OldProps}_{C^{i+1}}$, we need to show that $P \in \text{PotentialSuccessors}(C)$. Let $\texttt{rec}_k$ be a \texttt{reconfig} operation that calls $C^{i+1}.\text{CoS}$ while its flag $\texttt{firstTime}=false$. By Claim 11, $\texttt{rec}_k$ tracks $C$. Let $C^{m+1}$ be the biggest configuration in $\text{successors}^i(C)$ that is tracked by $\texttt{rec}_k$. Since $\texttt{rec}_k$ tracks configurations according to their sizes and since by (a) $C^{i+1}$ is the only configuration in $\text{successors}(C)$ of size is $i + 1$, $\texttt{rec}_k$ tracks $C^{i+1}$ immediately after it tracks $C^{m+1}$. By Observation 22, $\texttt{rec}_k$’s proposal in $C^{m+1}.\text{CoS}$ is included by some configuration in $\texttt{rec}_k$’s $\texttt{ToTrack}$ before $\texttt{rec}_k$ introduces $C^{i+1}$, and by definition, $\texttt{rec}_k$’s $\texttt{ToTrack}$ is included in $\text{PotentialSuccessors}(C^{m+1})$ before $\texttt{rec}_i$ introduces $C^{i+1}$. Now By the assumptions, $\text{PotentialSuccessors}(C^{m+1}) \subseteq \text{PotentialSuccessors}(C)$, and thus monotonic. Therefore, $P \in \text{PotentialSuccessors}(C^{m+1}) \subseteq \text{PotentialSuccessors}(C)$, and we are done.

\[\square\]

\textbf{Claim 13. Consider a nominated configuration $C$. Assume that no more configurations of size $|C|$ are introduced and $\{C\} \cup \text{PotentialSuccessors}(C)$ is monotonic. Denote $x \triangleq \max(\{|j| \mid \exists C^j \in \text{successors}^i(C) : |C^j| = j\})$. Then for every $|C| \leq i < x$:

- For every $C' \in \text{successors}^i(C)$, $\text{PotentialSuccessors}(C') \subseteq \text{PotentialSuccessors}(C)$

\textbf{Proof.} We prove by induction on $i$.

\textbf{Base:} $i = |C|$. If $i = x$, we are done. Otherwise, by the assumption, $C$ is the only introduced configuration of size $|C|$, and the lemma follows.

\textbf{Step:} Now assume that the lemma holds for some $|C| \leq i < x$, we show that it holds for $i + 1$. If $i + 1 \geq x$, we are done. If there is no configuration in $\text{successors}(C)$ whose size is $i + 1$, then (1) follows by induction. Otherwise, by the Claim 12 (a) and since $\text{PotentialSuccessors}(C)$ is monotonic, there is exactly one configuration $C^{i+1} \in \text{successors}^i(C)$ of size is $i + 1$. Since $i + 1 < x$, $C^{i+1}$ is not nominated. Therefore, whenever $C^{i+1}.\text{CoS}$ is called by a \texttt{reconfig} operation, its $\texttt{firstTime}=false$. Thus, all \texttt{reconfig} operations that call $C^{i+1}.\text{CoS}$ read $\texttt{startingPoint}=false$. Now let $C^{i+1} \in \text{PotentialSuccessors}(C^{i+1})$, we show that $C^{i+1} \in \text{PotentialSuccessors}(C)$. If $C^{n+1}$ is returned by $C^{i+1}.\text{CoS}$, then by Corollary 15, $C^{n+1} \in \text{OldProps}(C^{i+1})$. By Claim 12 (b), $\text{OldProps}(C^{i+1}) \subseteq \text{PotentialSuccessors}(C)$. Therefore, $C^{n+1} \in \text{PotentialSuccessors}(C)$, and we are done. Otherwise, there is a \texttt{reconfig} operation $\texttt{rec}$ that has $C^{n+1}$ in its $\texttt{ToTrack}$ before it invokes $C^{i+1}.\text{CoS}$. Thus $\texttt{rec}$'s first-
We now show that 
\[ C \]
in line 51 in the same iteration. Now consider two cases:

Consider another configuration \( C \).

Consider a configuration \( C \).

Claim 14. Consider two nominated configurations \( C, C' \) s.t. \( C = \text{Pred}(C') \), and assume that no more configurations with size \( |C| \) are introduced. If \( \{C\} \cup \text{PotentialSuccessors}(C) \) is monotonic, then (1) \( \text{successors}(C) \) is monotonic, (2) \( \text{OldProps}(C') \subseteq \text{PotentialSuccessors}(C) \), and (3) for every \( C'' \in \text{successors}(C'^{-1}) \), \( \text{PotentialSuccessors}(C'') \subseteq \text{PotentialSuccessors}(C) \).

Proof. Consider a configuration \( C_1 \in \text{PotentialSuccessors}(C') \), we start by showing that \( C_1 \supseteq C' \). By definition, there is a \text{reconfig} \( C_1 \) that calls \( C'.CoS \) in line 44 and \( C_1 \in \text{ToTrack} \) in line 51 in the same iteration. Now consider two cases:

1. \( C'.CoS \) in \( C_1 \) returns \( C_1 \). Therefore, by \( CoS \) property \( CoS_1 \), \( C_1 \supseteq C' \).

2. \( C_1 \) is in \( C_1 \)'s \text{ToTrack} before it invokes \( C'.CoS \). Thus, \( C_1 \) calls \( C'.CoS \) while its \text{firstTime} = false, and so by Claim 11, \( C_1 \) tracks \( C \). Now let \( C'' \) be the last configuration \( C_1 \) tracks before \( C' \), and note that \( C', C_1 \in \text{PotentialSuccessors}(C'') \). By the assumption, \( \text{PotentialSuccessors}(C'') \) is monotonic, and by the observation that \( C_1 \) tracks configurations from smallest to biggest, \( C_1 \supseteq C' \).

Consider another configuration \( C_2 \in \text{PotentialSuccessors}(C') \). By definition, there is a \text{reconfig} \( C_2 \) that calls \( C'.CoS \) in line 44 and \( C_2 \in \text{ToTrack} \) in line 51 in the same iteration. Now we show that \( C_1 \) and \( C_2 \) are related by containment. there are three cases:

1. Both \( C_1 \) and \( C_2 \) are returned by \( C'.CoS \). Therefore, by Claim 8, \( C_1 \) and \( C_2 \) are related by containment.

2. Neither \( C_1 \) nor \( C_2 \) is returned by \( C'.CoS \). Thus \( C_1 \) (\( C_2 \)) is in \( C_1 \)'s (respectively, \( C_2 \)'s) \text{ToTrack} before it invokes \( C'.CoS \). Thus, \( C_1 \) and \( C_2 \) call \( C'.CoS \) while their \text{firstTime} = false, and so by Claim 11, \( C_1 \) and \( C_2 \) track \( C \). Now let \( C_1' \) (\( C_2' \)) be the last configuration \( C_1 \) (respectively, \( C_2 \)) tracks before \( C' \), and note that \( C_1 \in \text{PotentialSuccessors}(C_1') \) (and \( C_2 \in \text{PotentialSuccessors}(C_2') \)). By the assumption, \( \text{PotentialSuccessors}(C_1') \), \( \text{PotentialSuccessors}(C_2') \subseteq S \), and thus \( C_1, C_2 \in S \). Now since \( S \) is monotonic, we are done.

3. One of the configurations, w.l.o.g. \( C_1 \) is returned by \( C'.CoS \), and \( C_2 \) is not. In this case, \( C_2 \) is in \( C_1 \)'s \text{ToTrack} before it invokes \( C'.CoS(P_2) \) and thus, by Observation
21, \( P_2 \supseteq C_2 \), and as in above \( C_2 \in S \). In addition, since \( rec_2 \) does not drop \( C_2 \) after \( C'.CoS \) returns, it reads \( startPoint=false \) during \( C'.CoS \). By Observation 24, there is a \( reconfig \) operation \( rec'_1 \) that gets \( C_1 \) from PreCompute during \( C'.CoS \). Now consider two cases:

(a) \( rec'_1 \) reads \( startPoint=false \). Therefore, \( rec'_1 \) calls \( C'.CoS \) while its \( firstTime=false \), and PreCompute returns its input which is therefore \( C_1 \). Thus, by definition, \( C_1 \in \text{OldProps}(C') \). By assumption, \( C_1 \in S \), and we are done.

(b) \( rec'_1 \) reads \( startPoint=true \). Since \( rec_2 \) reads \( startPoint=false \) during \( C'.CoS(P_2) \) and writes its proposal to \( Sarr \) (line 19) before reading \( startPoint \) 21, and since \( startPoint \) never changes from \( true \) false and \( rec'_1 \) finds it true, \( rec'_1 \) collects \( P_2 \) in \( Sarr \) in line 24. Therefore, \( C_1 \supseteq P_2 \supseteq C_2 \).

\[ \square \]

Claim 15. The set \( \text{PotentialSuccessors}(C_{init}) \cup \{C_{init}\} \) is monotonic.

Proof. By Corollary 16 Claim 11, all \( reconfig \) operations that calls \( C_{init}.CoS \) do so with \( firstTime=true \). Therefore, \( \text{OldProps}(C_{init}) = \{\} \) and all configurations in \( \text{PotentialSuccessors}(C_{init}) \) are returned from \( C_{init}.CoS \). Thus, by CoS property \( CoS_1 \), all configurations in \( \text{PotentialSuccessors}(C_{init}) \) contain \( C_{init} \), and by Claim 8, they are related by containment. The claim follows.

\[ \square \]

Lemma 48. The set of introduced configurations is monotonic.

Proof. Let \( x = |C_{init}| \). We will show by induction on \( i \geq x \) that the following are satisfied:

(a) The set \( \text{introducedSet}^i \) is monotonic.

(b) For every configuration \( C \in \text{nominatedSet}^i \), \( \{C\} \cup \text{PotentialSuccessors}(C) \) is monotonic.

The lemma will follow from (a). Base: we prove for \( i = x \). By Corollary 16, there is no introduced configuration other than \( C_{init} \) whose size is smaller than or equal to \( x \). Hence, (a) is satisfied; (b) follows from Claim 15.

Step: assume by induction that (a) and (b) hold for \( i \geq x \), we prove for \( i + 1 \). By Claim 9, every nominated configuration is also introduced, so if there is no introduced configuration whose size is \( i + 1 \), then we are done. Otherwise, let \( C \) be an introduced configuration s.t. \( |C| = i + 1 \), and let \( C_p = \text{Pred}(C) \). Note that by definition, \( C \in \text{successors}(C_p) \). Since \( |C| = i + 1 > \) \( x = |C_{init}| \), \( |C_p| < |C| = i + 1 \). Therefore, by the induction assumption (b), \( \{C_p\} \cup \text{PotentialSuccessors}(C_p) \) is monotonic, and by the induction assumption (a), \( C_p \) is the only introduced configuration of size \( |C_p| \).

(a): By the first induction assumption \( \text{introducedSet}^{|C|} \) is monotonic, thus it is enough to show that \( C \) contains or equals every configuration \( C' \in \text{introducedSet} \) s.t. \( |C'| \leq |C'| \leq i + 1 \). Note that, by definition, \( C' \in \text{successors}(C_p) \). By Corollary 18 (1), \( \text{successors}(C_p) \)

```
is monotonic. Therefore, \( C \) and \( C' \) are related by containment, and since \( |C| = i + 1 \geq |C'| \), \( C \) contains or equals \( C' \), as requested.

\( \textbf{(b)} \): By (a), \( C \) is the only introduced configuration whose size is \( i + 1 \). If \( C \) is not nominated, then we are done. Otherwise, let \( S = \text{PotentialSuccessors}(C_p) \) by Corollary 18 (2), \( \text{OldProps}(C) \subseteq S \) and by 18 (3), for every \( C'' \in \text{successors}|C|^{-1}(C_p) \), \( \text{PotentialSuccessors}(C'') \subseteq S \). Therefore, by Claim 14, \( \{C\} \cup \text{PotentialSuccessors}(C) \) is monotonic, as needed.

We are now ready to conclude the complexity of the dynamic objects that uses our algorithm, which is captured by the following lemma:

**Lemma 49.** Consider an execution of a dynamic objects that uses our algorithm, and consider a loop in which Check\((C)\) is repeatedly called, s.t. \( C \) is the configuration returned from the previous Check, until some Check\((C')\) returns \( \langle C', \ast \rangle \). Let \( n \) be the number of Propose\((P)\) operations in the execution. Then:

1. All the Checks in the loop (together) return \( O(n) \) configurations in the speculations sets.
2. The complexity of all Checks in the loop combined is \( O(n) \).

**Proof.** Since every Check in the loop starts where the previous returns, the Checks in the loop introduce different configurations. Thus by Lemma 48, we immediately conclude that the number of configurations returned in speculated sets of all Checks in the loop together is bounded by \( n \). Moreover, by CoS1, no configuration is returned more than once in the speculation sets. It is left to show that the complexity of all Checks combined is \( O(n) \). First observe (again by Lemma 48) that all Checks combined invokes at most \( n \) CoSs. Second, each CoS writes at most three times to shared registers (lines 10, 18, and 19), reads once (in line 21), and performs each of the collects in lines 11, 15, and 24 at most once.

Now observe that CoS performs the collect in line 27 only if the previous collect (in line 24 or 27) contained a proposal \( P_1 \not\subseteq P \), which means that none of the CoSs collected \( P_1 \) before. Since there are at most \( n \) proposals, all CoSs together perform the collect in line 27 at most \( n \) times. All in all, we get that the complexity of all Checks in the loop is \( O(n) \).

---

5.6 Conclusions

In this chapter we defined a dynamic model with a clean failure condition that allows an administrator to reconfigure an object and switch a removed server off once the reconfiguration operation completes. In this model, we have captured a succinct abstraction for consensus-less reconfiguration, which dynamic objects like atomic read/write register
and max-register may use. We demonstrated the power of our abstraction by providing an optimal implementation of a dynamic register, which has better complexity than previous solutions in the same model.
Chapter 6

Conclusion

In this thesis we studied asynchronous distributed reliable storage and focused on two fundamental aspects that are essential in every distributed storage system: space cost and dynamic reconfiguration.

Storage space cost. The most common approach to achieve reliable distributed storage is via replication, i.e., storing multiple copies of each data block (on different servers). In this thesis we first compared the fundamental space cost of such algorithms as a function of the basic primitives they use (e.g., read/write registers versus max-registers). We introduced a new hierarchy, which classifies primitive types by the number of base objects of a given primitive required to emulate an $f$-tolerant register, as a function of the number of writers $k$ and the number of available servers $n$.

Then, we considered algorithms that try to mitigate the significant cost of replication, which results from the immense size of the data, by using a symmetric black-box coding scheme. Given three problem parameters: $f, c,$ and $D$, where $f$ is the number of storage node failures tolerated, $c$ is the concurrency allowed by the algorithm, and $D$ is the data size, we proved that the storage cost is $\Theta(\min(f, c) \cdot D)$. Asymptotically, this means either a storage cost as high as that of replication, or as high as keeping as many versions of the data as the concurrency level. Then, equipped with the insights from the lower bound, we also presented an algorithm that combines replication and erasure codes, whose storage cost is $O(\min(f, c) \cdot D)$.

Dynamic reconfiguration. In this thesis we contributed to the understanding of reconfiguration of distributed storage by showing both negative (impossibility) and positive (algorithms) results. We first proved that in an asynchronous API-based reconfigurable model allowing at least one failure, without restricting the number of reconfigurations, there is no way to emulate dynamic safe wait-free storage. We further showed how to circumvent this result using a dynamic eventually perfect failure detector the we defined: we presented an algorithm that uses such a failure detector in order to emulate a
wait-free dynamic atomic MWMR register.

Then, we defined a dynamic API-based model with a clean failure condition that allows an administrator to reconfigure an object and switch a removed server off once the reconfiguration operation completes. In this model, we have captured a succinct abstraction for consensus-less reconfiguration, which dynamic objects like atomic read/write register and max-register may use. We demonstrated the power of our abstraction by providing an optimal implementation of a dynamic register, which has better complexity than previous solutions in the same model.
Bibliography


.1 Additional results of Chapter 2

We prove here an extended version of Lemma 2 and use it to show additional lower bounds (see Theorems 12, 13, 14, and 15).

**Lemma 2** (extended). For all $k > 0$, $f > 0$, let $A$ be an $f$-tolerant algorithm that emulates a WS-Safe obstruction-free $k$-register using a collection $S$ of servers storing a collection $B$ of wait-free MWMR atomic registers. Then, for every $F \subseteq S$ such that $|F| = f + 1$, there exist $k$ failure-free runs $r_i$, $0 \leq i \leq k$, of $A$ such that (1) $r_0$ is a run consisting of an initial configuration and $t_0 = 0$ steps, and (2) for all $i \in [k]$, $r_i$ is a write-only sequential extension of $r_{i-1}$ ending at time $t_i > 0$ that consists of $i$ complete high-level writes of $i$ distinct values $v_1, \ldots, v_i$ by $i$ distinct clients $c_1, \ldots, c_i$ such that:

- $|\text{Cov}(t_i)| \geq if$
- $|\delta(\text{Cov}(t_i)) \cap F| = \emptyset$
- $|\delta(\text{Tr}(t_i) \setminus \text{Cov}(t_{i-1}))| > 2f$
- $\text{Cov}(t_i) \supseteq \text{Cov}(t_{i-1})$
- $|\text{Cov}(t_i)| \geq f$
- $|\delta(\text{Cov}(t_i) \setminus \text{Cov}(t_{i-1}))| \geq f$
- $|\delta(\text{Tr}(t_i) \setminus \text{Cov}(t_{i-1}))| \geq f$
- $\text{Cov}(t_i) \supseteq \text{Cov}(t_{i-1})$

**Proof.** Fix arbitrary $k > 0$, $f > 0$, and a set $F$ of servers such that $|F| = f + 1$. We proceed by induction on $i$, $0 \leq i \leq k$. **Base:** Trivially holds for the run $r_0$ of $A$ consisting of $t_0 = 0$ steps. **Step:** Assume that $r_{i-1}$ exists for all $i \in [k-1]$. We use the extension $r'$ constructed in the proof of Lemma 2 in Section 2.3 to show the implications $c) - e)$ of the extended version are true:

- $|\delta(\text{Tr}(t'_i) \setminus \text{Cov}(t_{i-1}))| > 2f$: Follows immediately from Lemma 5 and Definition 2.1.
- $|\delta(\text{Cov}(t_i)) \setminus \text{Cov}(t_{i-1}))| \geq f$: Since by Corollary 2, $|Q_i(t_i)| = f$, and by Lemma 3.2, $Q_i(t_i) \subseteq Q_i(t'_i)$, we get $|Q_i(t'_i)| = f$. Hence, by Definition 2.4, $|\delta(\text{Cov}(t'_i))| \geq |Q_i(t'_i)| \geq f$, and by Definition 2.3, $|\delta(\text{Cov}(t_i) \setminus \text{Cov}(t_{i-1}))| = |\delta(\text{Cov}(t'_i))| \geq f$.
- $\text{Cov}(t_i) \supseteq \text{Cov}(t_{i-1})$: Follows immediately from Definition 4.

**Theorem 12.** For all $k > 0$, and $f > 0$, let $A$ be an $f$-tolerant algorithm emulating a WS-Safe obstruction-free $k$-register using a collection $S$ of servers. Then, $|S| \geq 2f + 1$.

**Proof.** By Lemma 2.c), there exists a run $r_1$ of $A$ consisting of $t_1$ steps such that $|\delta(\text{Tr}(t_1) \setminus \text{Cov}(0))| > 2f$. Therefore, $|S| \geq |\delta(\text{Tr}(t_1))| \geq 2f + 1$.

**Number of registers per server.** Theorem 2 implies that in case $|S| = 2f + 1$, at least $(2f + 1)k$ registers are required. The following theorem further refines this result by showing that in this case, each server must store at least $k$ registers:
Theorem 13. Let $|S| = 2f + 1$. For all $k > 0$, $f > 0$, every $f$-tolerant algorithm emulating an obstruction-free WS-Safe $k$-register stores at least $k$ registers on each server in $S$.

Proof. Pick an arbitrary $f > 0$, $k > 0$, and suppose toward a contradiction that there is an $f$-tolerant algorithm $A$ emulating an obstruction-free WS-Safe $k$-register that stores less than $k$ registers on some server $s \in S$ (i.e., $|\delta^{-1}([s])| < k$).

Pick an arbitrary set $F \subset S$ of size $|F| = f + 1$ such that $s \notin F$. By Lemma 2, there exist a sequential write-only run $r_k$ consisting of $k$ high-level write invocations $W_1, \ldots, W_k$ by $k$ distinct clients, and $k$ distinct times $t_1 < \cdots < t_k$ such that: $|\delta(Cov(t_1))| \geq f$ and $\delta(Cov(t_1)) \cap F = \emptyset$; and for all $i \in [k] \setminus \{1\}$, $|\delta(Cov(t_i) \setminus Cov(t_{i-1}))| \geq f$, $\text{Cov}(t_i) \supset \text{Cov}(t_{i-1})$, and $\delta(Cov(t_i)) \cap F = \emptyset$. By induction on $i \in [k]$, it is easy to see that all sets in the collection consisting of $\text{Cov}(t_1)$ and $\text{Cov}(t_i) \setminus \text{Cov}(t_{i-1})$ where $i \in [k] \setminus \{1\}$ are pairwise disjoint. Thus, at least $f$ new registers become covered at each $t_i$, $i \in [k]$. Moreover, since no registers on the servers in $F$ are covered at $t_i$, all registers that become covered at $t_i$ must be located on the servers in $S \setminus F$. Therefore, since $|S \setminus F| = f$ and $s \in S \setminus F$, we conclude that at least $k$ distinct registers on $s$ must be covered at time $t_k$, that is, $|\delta^{-1}([s]) \cap \text{Cov}(t_k)| \geq k$. Therefore, $|\delta^{-1}([s])| \geq k$. A contradiction. \hfill $\Box$

Servers with bounded storage. The following result provides a lower bound on the number of servers for the case the storage available on each server is bounded (as it is often the case in practice) by a known constant:

Theorem 14. Let $m > 0$ be an upper bound on the number of registers mapped to each server in $S$ (i.e., $\forall s \in S, |\delta^{-1}([s])| \leq m$). For all $f > 0$ and $k > 0$, every $f$-tolerant algorithm emulating an obstruction-free WS-Safe $k$-register from a collection $S$ of servers such that $|S| > 2f + 1$ uses at least $\lceil kf/m \rceil + f + 1$ servers (i.e., $|S| \geq \lceil kf/m \rceil + f + 1$).

Proof. Fix $F \subset S$ such that $|F| = f + 1$. By Lemma 2(a), there exists an extension $r_k$ of $r_{k-1}$ ending at time $t_k$ such that $|\delta(T_r(t_k) \setminus \text{Cov}(t_{k-1}))| > 2f$. Since $|F| = f + 1$, $|\delta(T_r(t_k) \setminus \text{Cov}(t_{k-1})) \cap F| \leq f + 1$. Hence, we receive

$$|\delta(T_r(t_k) \setminus \text{Cov}(t_{k-1})) \setminus F| =$$

$$|\delta(T_r(t_k) \setminus \text{Cov}(t_{k-1}))| - |\delta(T_r(t_k) \setminus \text{Cov}(t_{k-1})) \cap F| \geq 2f + 1 - f - 1 = f$$

Thus,

$$|(T_r(t_k) \setminus \text{Cov}(t_{k-1})) \setminus \delta^{-1}(F)| \geq |\delta(T_r(t_k) \setminus \text{Cov}(t_{k-1})) \setminus F| \geq f \quad (1)$$

On the other hand,

$$|(T_r(t_k) \setminus \text{Cov}(t_{k-1})) \setminus \delta^{-1}(F)| = |(T_r(t_k) \setminus \text{Cov}(t_{k-1})) \cap \delta^{-1}(S \setminus F)| =$$

$$|(T_r(t_k) \cap \delta^{-1}(S \setminus F)) \setminus \text{Cov}(t_{k-1})| \leq |\delta^{-1}(S \setminus F) \setminus \text{Cov}(t_{k-1})| \quad (2)$$
Since by Lemma 2(b) and (d), $|\text{Cov}(t_{k-1})| \geq (k - 1)f$ and $\delta^{-1}(S \setminus F) \supseteq \text{Cov}(t_{k-1})$, we get

$$|\delta^{-1}(S \setminus F) \setminus \text{Cov}(t_{k-1})| = |\delta^{-1}(S \setminus F)| - |\text{Cov}(t_{k-1})| \leq (|S \setminus F|)m - (k - 1)f \quad (3)$$

Combining (3) with (1) and (2), we get

$$(|S \setminus F|)m - (k - 1)f \geq |(T_r(t_k) \setminus \text{Cov}(t_{k-1})) \setminus \delta^{-1}(F)| \geq f$$

Since $S \supseteq F$ and $|F| = f + 1$, we obtain $(|S \setminus F|)m - (k - 1)f = |S|m - (f + 1)m - (k - 1)f \geq f$. Therefore, $|S|m \geq fm + m + kf$, which implies that $|S| \geq kf/m + f + 1$. Since $|S|$ is an integer, we conclude that $|S| \geq \lceil kf/m \rceil + f + 1$. \hfill \Box

**Adaptivity to Contention**  
Given a run fragment $r$ of an emulation algorithm, the point contention $[2, 13]$ of $r$, $\text{PntCont}(r)$, is the maximum number of clients that have an incomplete high-level invocation after some finite prefix of $r$. Similarly, we use $\text{PntCont}(op)$ to denote $\text{PntCont}(r_{op})$, where $r_{op}$ is the run fragment including all events between the $op$’s invocation and response.

The resource complexity of $A$ is **adaptive to point contention** if there exists a function $M$ such that after all finite runs $r$ of $A$, the resource consumption of $A$ in $r$ is bounded by $M(\text{PntCont}(r))$. Likewise, the time complexity of $A$ is **adaptive to point contention** if there exists a function $T$ such that for each client $c_i$ and operation $op$, the time to complete the invocation of $op$ by $c_i$ is bounded by $T(\text{PntCont}(op))$.

We show that no WS-Safe obstruction-free MWSR register can have a fault-tolerant emulation adaptive to point contention:

**Theorem 15.** For any $f > 0$, there is no $f$-tolerant algorithm that emulates an WS-Safe obstruction-free $k$-register with resource complexity adaptive to point contention.

**Proof.** By Lemma 2, there exists a run $r$ of $A$ consisting of $k$ high-level writes by $k$ distinct clients such that the resource complexity grows by $f$ for each consecutive write that completes in $r$ whereas the point contention remains equal 1 for the entire $r$. We conclude that no function mapping point contention to resource consumption can exist, and therefore, $A$’s resource complexity is not adaptive to point contention. \hfill \Box
Hebrew Section
שנחו את חסומת הב الطلבים על ידי חסום חסום ונדרים לchers, אלגוריתם חסום.

באת החזון והרכז הרצוח לchers, אלגוריתם חסום.

את החזון והרכז הרצוח לchers, אלגוריתם חסום.

את החזון והרכז הרצوح לchers, אלגוריתם חסום.

את החזון והרכז הרצוח לchers, אלגוריתם חסום.

ברخش כמו כן, ונדרים לchers, אלגוריתם חסום.
המחקרakening בענワイיה פרופ' עדית קדר בפקולטה להנדסת חשמל על שם יוטרייב טכנון – מכון טכנולוגי לישראל.

תודה נוגה לテכניון – מכון טכנולוגי לישראל ולקרן עזריאלי על התמיכה הכספית בהשתלמותי.
אלגוריתמים וחסמים במערכות אחסון מובחרות

תחודש על מתקד

לשמ מילוי חלקי של הדרישות ל😂בקת התוכן

דוקטור לפילוסופיה

אלכסנדר שפיגלמן

אלכסנדר שפיגלמן

זוהב לסנ הנטכינז – מוכזז טככליי ליישאר

סינ התשע"ח חיפה י"י 2018