

The Hidden Complexity of Distributed Systems

Présentée le 15 octobre 2020

à la Faculté informatique et communications
Laboratoire de calcul distribué
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Karolos ANTONIADIS

Acceptée sur proposition du jury

Prof. E. Telatar, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. R. Friedman, rapporteur
Prof. F. Pedone, rapporteur
Prof. M. Grossglauser, rapporteur

Τό νά ὁραματίζεσαι ἕνα σκοπό, αὐτό δέν εἶναι ἀντίθετο μέ τήν πράξη, ἀπεναντίας εἶναι ἡ ἀρχή της. Ἀλλά πρέπει ἐν συνεχείᾳ νά ὀρίσεις τα μέσα τοῦ σκοποῦ, τά συγκεκριμένα σκαλοπάτια ἕνα ἕνα καί νά τά παῖς. Ἐδῶ ὁμως πιά ὁ ἡδονισμός τοῦ ωραίου καί τοῦ ὑψηλοῦ σέ ἐγκαταλείπει καί θᾶπρεπε νά τόν ἀντικαταστήσει ἡ πειθαρχία καί ἡ ἄσκηση τῆς συγκέντρωσης καί τῆς δουλειᾶς.
— Κωνσταντῖνος Τσάτσος

Man muss nur gehn: Kein Gefühl ist das fernste.
— Rainer Maria Rilke

To my parents,
Sigrid and Jannis

Acknowledgements

I am extremely grateful to Prof. Rachid Guerraoui for giving me the opportunity to freely pursue the work presented in this thesis. I want to thank him for being a great mentor, extremely patient, and encouraging during all those years.

I was lucky enough to meet some amazing people during my studies that I have the honor of considering good friends: Georgios Chatzopoulos, Georgios Damaskinos, David Kozhaya, Dragos-Adrian Seredinschi, Vasileios Trigonakis, and Igor Zablotchi. Special thanks go to Igor Zablotchi: without our discussions this work would not have been possible. Additionally, the whole PhD experience would have been rather dull without the interaction with Aggelos Biboudis, Panayiotis Danassis, Tudor David, Chi Thang Duong, El Mahdi El Mhamdi, Jad Hamza, Lê Nguyễn Hoàng, Christos Kotsalos, Rhicheck Patra, Matej Pavlovic, Javier Picorel, Georg Stefan Schmid, Mahsa Taziki, and Jingjing Wang. Furthermore, I would like to thank France Faille and Fabien Salvi for always being there to help and answer my bureaucratic, and not only questions.

I would like to express my gratitude to two persons that played a great role in my academic path. I owe a lot to Prof. Panagiota Fatourou at University of Crete for inspiring me in my early study years and introducing me to research, and to Prof. Souzana Papadopoulou for being a lifelong mathematics mentor and a great role model.

Naturally, I owe everything to my parents, Sigrid and Jannis, and my siblings Katerina and Antonios, for always being there for me, no matter what. Words cannot describe how thankful and fortunate I am to be part of this great family.

Finally, I want to thank my wife, Maria. Meeting her is the best and most beautiful thing that ever happened to me. I am glad she was part of this journey. The best is yet to come.

Bern, September 11, 2020

K. A.

Preface

The work presented in this thesis was conducted in the Distributed Computing Laboratory at EPFL under the supervision of Prof. Rachid Guerraoui.

The results of this thesis are based on the following articles:

- Karolos Antoniadis, Vincent Gramoli, Rachid Guerraoui, Eric Ruppert, and Igor Zablotchi. “*Leaderless Consensus.*” Under submission.
- Karolos Antoniadis, Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel. “*The Impossibility of Fast Transactions.*” In the Proceedings of the 34th International Parallel and Distributed Processing Symposium (IPDPS), 2020.
- Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. “*State Machine Replication is More Expensive than Consensus.*” In the 32nd International Symposium on Distributed Computing (DISC), 2018.

Despite the aforementioned papers, during my doctoral studies, I was also involved in work that resulted in the following publications:

- Karolos Antoniadis, Rachid Guerraoui, and Vasileios Trigonakis. “*Thread-Placement Learning.*” In the Proceedings of the 40th International Conference on Distributed Computing Systems (ICDCS), 2020.
- (Book Chapter) Karolos Antoniadis and Rachid Guerraoui. “*The Notions of Time and Global State in a Distributed System.*” In the book “*Concurrency: The Works of Leslie Lamport*”, 2019.
- Karolos Antoniadis, Peva Blanchard, Rachid Guerraoui, and Julien Stainer. “*The entropy of a distributed computation random number generation from memory interleaving.*” In the Distributed Computing Journal, 2018.

Abstract

The field of distributed computing has a long history, of more than fifty years. During that time, our understanding of the field has improved immensely and a certain body of folklore beliefs has formed. However, such folklore beliefs are not necessarily always true.

In this thesis, we identify hidden complexities (in the sense of intricacies or costs) of distributed systems that contradict some of these folklore beliefs. Specifically, in this thesis, we challenge accepted beliefs in the following way: i) consensus algorithms need not be leader-based: there exists a deterministic leaderless consensus algorithm that is robust to non-synchronous periods, ii) completing a state machine replication command can be arbitrarily more expensive than solving a consensus instance, and iii) no data store actually provides fast transactions because they are impossible. Our results are associated to some of the fundamental problems in the field of distributed computing and are of significant practical relevance.

Keywords: consensus, fast transactions, leaderless, state machine replication

Zusammenfassung

Der Bereich des verteilten Rechnens hat eine lange Geschichte von mehr als fünfzig Jahren. In dieser Zeit hat sich unser Verständnis auf diesem Gebiet immens verbessert, und es hat sich ein gewisser “Volksglaube” herausgebildet. Solche Überlieferungen sind jedoch nicht unbedingt immer wahr.

In dieser Arbeit identifizieren wir versteckte Komplexitäten (im Sinne von Feinheiten oder Kosten) verteilter Systeme, die einigen dieser überlieferten Annahmen widersprechen. Konkret stellen wir in dieser Arbeit akzeptierte Überzeugungen auf folgende Weise in Frage: i) Konsensusalgorithmen müssen nicht führerbasiert sein: Es gibt einen deterministischen führerlosen Konsensusalgorithmus, der robust gegenüber nicht-synchronen Perioden ist, ii) die Ausführung einer Replikation von Zustandsautomaten kann beliebig teurer sein als die Lösung einer Konsensusinstanz, und iii) kein Datenspeicher bietet tatsächlich schnelle Transaktionen, weil sie unmöglich sind. Unsere Ergebnisse stehen im Zusammenhang mit einigen der grundlegenden Probleme auf dem Gebiet des verteilten Rechnens und sind von erheblicher praktischer Relevanz.

Schlüsselwörter: Konsensus, schnelle Transaktionen, führerlos, Replikation von Zustandsautomaten

Contents

Acknowledgements	i
Preface	iii
Abstract (English/Deutsch)	v
List of Figures	xi
1 Introduction	1
1.1 Distributed Computing Models	2
1.2 Consensus and State Machine Replication	3
1.3 Transactions	3
1.4 Contributions	4
1.5 Roadmap	6
I Consensus and State Machine Replication	7
2 Leaderless Consensus	9
2.1 Introduction	9
2.2 Model	11
2.3 Leaderless Termination	14
2.4 Archipelago: Leaderless Consensus	17
2.5 Archipelago: Proof of Correctness	21
2.6 Leaderless Consensus in Message Passing	31
2.7 ArchSMR: Archipelago in Practice	37
2.8 Related Work	40
2.9 Conclusion	41
3 State Machine Replication is More Expensive Than Consensus	43
3.1 Introduction	43
3.2 Model	46
3.2.1 Consensus	47
	ix

Contents

3.2.2	State Machine Replication	49
3.3	Complexity Lower Bound on State Machine Replication	58
3.3.1	Complexity Lower Bound	58
3.3.2	Extension to other Models	60
3.4	The Empirical Perspective	61
3.4.1	Experimental Methodology	61
3.4.2	Experimental Results on a Single Machine	63
3.4.3	Wide-area Experiments	65
3.5	Discussion	67
3.6	Conclusion	68
II	Transactions	71
4	The Impossibility of Fast Transactions	73
4.1	Introduction	73
4.2	Model	75
4.3	Fast Transactions Are Impossible	85
4.4	Unbounded-Version Data Store	90
4.5	Conclusion	100
5	Concluding Remarks	103
	Bibliography	105
	Curriculum Vitae	113

List of Figures

2.1	Graphical depiction of a synchronous-1 execution.	13
2.2	With 2 processes, Archipelago might never decide in a synchronous-1 execution ($v' > v$).	23
2.3	Execution pattern that appears when the minimum value propagates to the next adopt-commit-max object ($x \geq 2$).	25
2.4	Lemma 7 (1)	29
2.5	Lemma 7 (2)	30
2.6	Lemma 7 (3)	30
2.7	Performance of ZooKeeper and ArchSMR upon server suspension.	39
2.8	Performance of ArchSMR upon rotating suspensions of all servers.	40
3.1	Constructed execution of Theorem 4. Red dashed lines correspond to rounds where a replica is suspended. Replica p_1 is suspended for a_1 rounds, replica p_2 for a_2 rounds, etc.	59
3.2	Experimental results with LibPaxos on a single-machine setup. We compare the cost of SMR commands with the cost of consensus instances in three scenarios.	63
3.3	Experimental results with LibPaxos on the WAN. Similar to Figure 3.2, we compare the cost of SMR commands with the cost of consensus instances.	65
3.4	Experimental results with Raft on the WAN. Similar to Figures 3.2 and 3.3, we compare the cost of SMR commands with the cost of consensus instances.	66
4.1	A transaction t that reads, among others, objects o and o' cannot read values v_i and v_j due to the existence of e_w . However, transaction t can read values v and v_j for objects o and o' respectively. Note that all three writes e_{w_i} , e_w , and e_{w_j} are performed by the same client c	84
4.2	Executions α_1 and α_2 where client c_w writes (in red) objects o_1 and o_2 . Client c_h issues a finite number of transactions (in green) between the c_w 's writes and c_r performs a transaction that reads both objects o_1 and o_2 (in blue).	86

- 4.3 At the top, we depict execution α where client c_w alternates between writing (in red) objects o_1 and o_2 . Client c_h issues a finite number of transactions (in green) after client's c_w write of value $v_{o_2}^j$ until values $(v_{o_1}^j, v_{o_2}^j)$ are visible. At the bottom, we depict execution α_i . Due to space constraints, we depict the transactions of c_h until values $(v_{o_1}^i, v_{o_2}^i)$ and $(v_{o_1}^{k+2}, v_{o_2}^{k+2})$ are visible with shortened green boxes. The events of client's c_r transaction that read objects o_1 and o_2 are depicted in blue. 88

1 Introduction

Since the completion of the first fully automatic digital computer in 1941 [126], computers have been fundamentally enhanced in at least two ways. First, the fundamental work of ARPANET in the sixties [88] has led to computer networks, making computers substantially more practical and subsequently changing the world as we know it. Second, with the advent of multicores and multiprocessors, computers have now multiple processing units, and hence the ability to perform computations in parallel that allowed for groundbreaking new technologies to emerge. Both of these fundamental enhancements brought distributed systems to the foreground of computing.

A *distributed system* is intuitively a set of independent *processes* coordinating with each other to achieve a common goal. One might think of a “distributed system” as a set of geographically distant computers that achieve a specific goal by communicating over a network (e.g., Internet). However, it is not necessary that the processes are geographically distant. Based on the above intuitive definition, a multi-threaded program running on top of a multicore can be characterized as a distributed system.

Distributed systems can provide us with a few key benefits, such as speed and robustness. Speed, because we can use multiple resources to solve a problem. For example, a multi-threaded program can take advantage of the machine’s underlying cores and hence complete its tasks faster. Also, we can reduce a user’s experienced latency, by sharing, for example the resources of an application in a such a way so that the the user’s desired resources are geographically closer to the user. Robust, because even if some process crashes, the system can still take advantage of the remaining non-crashed processes to perform computations. Distributed systems are the cornerstone of today’s Internet services, such as search engines, social networks, banking, etc. However, distributed systems are also found in industrial settings [101], or even the Boeing 777 aircraft system [124].

The field of *distributed computing* studies distributed systems, and has a long history of more than fifty years starting from the fundamental work of pioneers such as Edsger W. Dijkstra that investigated the mutual exclusion problem [46]. Distributed computing examines whether a problem is solvable in a distributed system, as well as the efficiency of such a solution. Although distributed computing can be practical in nature, in this thesis and unless said otherwise, when we talk about distributed computing we refer to theoretical distributed computing. Specifically, distributed computing concerns mathematical *models* that capture the characteristics of real or theoretical distributed systems.

In the rest of this chapter, we describe some preliminary notions behind models used in distributed systems. Then, we present the fundamental abstractions of consensus, state machine replication, and of a transaction in a distributed system before listing the contributions of this thesis. We conclude this chapter by presenting the roadmap for the rest of this dissertation.

1.1 Distributed Computing Models

To study a distributed system, we need to abstract the underlying distributed system by devising a mathematical model. There are two broad categories of models that we typically use, depending on whether we are modeling a message-passing (i.e., processes communicate over a network) or a shared-memory (e.g., processes communicate by sharing memory) distributed system. For example, we can model a message-passing *distributed system* as a set of state machines where each state machine corresponds to a process and these state machines change their state in response to a received message or of some local computation.

A model should be as general as possible but should also try to capture the peculiarities of the modeled distributed system. Therefore, we can further classify a message-passing or shared-memory model as synchronous or asynchronous. For example, in an embedded processor we may assume that there are bounds on the time it takes for a process to read from or write to shared memory, and those bounds are strictly enforced. Similarly, in an industrial setting we may assume that there are bounds on the time it takes for a message to be transmitted to another process. A model that assumes such bounds is called *synchronous*. However, a synchronous model would not be able to capture a general multicore machine, because a long context switch might violate these timing assumptions. Similarly, modeling a distributed system running on top of a wide-area network with a synchronous model is a bad idea, because the transmission of a message over the network might take an arbitrary amount of time. We can therefore devise a model that makes no such timing assumptions. We call such a model *asynchronous*. We say that a model is *eventually synchronous* when there is a point after which it behaves synchronously (i.e., some timing assumptions eventually hold).

Having a model, we can formally answer on whether we can solve a specific problem by using

the underlying (modeled) distributed system, and how efficient such a solution is. Solving a problem is equivalent to devising an algorithm, namely defining the set of state machines.

Finally, we can further augment a model by capturing the type of failures that can occur in the underlying distributed system. For example, a machine might crash, some process might behave in a malicious way, there might be a network partition in the network, etc. We say that an algorithm is *fault-tolerant* if the algorithm operates in a model that allows for some of the processes to be faulty (e.g., crashed).

1.2 Consensus and State Machine Replication

The problem of *consensus*, formally introduced by Pease, Shostak, and Lamport [103] is one of the fundamental problems in the field of distributed computing. In this problem, a set of distributed processes need to reach agreement on a single value. Consensus has been extensively studied with fundamental results such as FLP [51] and also plays an important role in shared-memory systems [61].

Agreeing only once on something is arguably not that useful in a distributed system. However, by utilizing multiple instances of consensus, we can create what is known as *state machine replication* (SMR). Essentially, SMR consists of replicating a sequence of commands – often known as a log – on a set of processes which replicate the same state machine. These commands represent the ordered input to the state machine. SMR can be used to transform almost any algorithm into a fault-tolerant one. This is achieved by replicating the same state machine across multiple servers. When one crashes, the remaining servers remain accessible, giving the illusion of a single always available machine.

SMR has been successfully deployed in applications ranging from storage systems [102], to lock and coordination services [64]. At a high level, SMR can be viewed as a sequence of consensus instances, so that each value output from an instance corresponds to a command in the SMR log.

1.3 Transactions

A *transaction* is a fundamental abstraction for manipulating multiple objects atomically in a data store. Data stores can exhibit erratic behaviour for a plethora of reasons: computers might crash, there could be bugs in the code of the database, etc. [69]. Transactions ameliorate the negative effects of these problems, as well as aid the programmer's job. For this reason, transactions are now ubiquitous in the world of data stores, and most of the well-known database systems provide them (e.g., MySQL, PostgreSQL, and MongoDB).

The most basic interface of a data store is a read-write interface [74] on a set of objects, where the objects can be identified by what is called a key. At a high-level, the client issues a request on the data store by identifying the object (e.g., by providing the key) the client wants to access. Subsequently, the server responds back to the client with the desired data. Data stores augment their interface by providing transactions. Data stores being extensively used for read-heavy workloads, aim to optimize read-only transactions since they are the most frequent in practice [28]. It is thus natural to seek implementations for read-only transactions that are as fast as possible.

Lately, there has been an attempt to introduce the notion of a *fast transaction* [87]. A fast transaction captures the fact that a read-only transaction is one round-trip, non-blocking, and one-version. *One round-trip* means that a client does not contact a server more than once during a transaction. *Non-blocking* states that servers should not communicate with each other before responding to the client. Finally, *one-version* asks that a server only sends one value for each read object.

1.4 Contributions

The field of distributed computing has a long and rich history. During its long history, distributed computing produced an impressive battery of theoretical results with which we can understand the field very well. Thus, for better or for worse, a certain body of folklore beliefs, has formed. However, these folklore beliefs are not necessarily always true.

Oxford English Dictionary defines “complexity” as “the state or quality of being intricate or complicated.” Contemporary distributed systems exhibit a level of complexity that sometimes refute the formed beliefs.

In this thesis, we uncover hidden complexities (in the sense of intricacies or costs) of distributed systems that challenge accepted beliefs in the following way: i) consensus algorithms need not be leader-based: there exists a deterministic leaderless consensus algorithm that is robust to non-synchronous periods, ii) completing a state machine replication command can be arbitrarily more expensive than solving a consensus instance, and iii) no data store actually provides fast transactions because they are impossible. In what follows, we present three flavors of intricacies and present our concrete contributions.

Consensus Classic synchronous consensus algorithms are leader-based. A leader helps them converge fast when the system is initially synchronous, but impacts performance when the system is not. Therefore, a leader-based algorithm has to pay a substantial performance cost when the leader is slow or crashes. In other words, leader-based consensus algorithms are not robust to non-synchronous periods of time. This thesis asks whether,

under eventual synchrony, it is possible to deterministically solve consensus without a leader. Someone might believe that some form of leader is inherently needed, because the weakest failure detector to solve consensus provides a leader [37]. We prove that this is not the case.

Specifically, we:

1. study the very definition of a leaderless consensus algorithm;
2. devise and prove correct an eventually synchronous deterministic leaderless consensus algorithm;
3. utilize our leaderless consensus algorithm to build a state machine replication algorithm and illustrate its robustness.

State Machine Replication (SMR) Consensus and SMR are generally considered to be equivalent problems. Indeed, the two problems are computationally equivalent: any solution to the former problem leads to a solution to the latter, and vice versa. In this thesis, we study the relation between consensus and SMR from a complexity perspective. We find that, surprisingly, completing an SMR command can be more costly than solving a consensus instance.

Our contributions are:

1. proving that completing an SMR command is more expensive than solving a consensus instance (i.e., in a model where each consensus instance takes bounded time to complete, completing an SMR command might take an arbitrary amount of time);
2. empirically showing that our result corresponds to practical phenomena.

Transactions We prove that fast transactions in a fault-tolerant data store have to perform a write and argue that a transaction that writes cannot be non-blocking. This by definition contradicts the very existence of fast transactions and renders them impossible. Specifically, we show that fast transactions are impossible if we want to tolerate the failure of even one server. In other words, by unveiling the hidden cost (i.e., writing) of fast transactions, we refute the belief that fast transactions are possible.

In this thesis, we:

1. devise a formal framework that is general enough to capture any data store, while at the same time precisely captures the notion of fast read-only transactions;
2. prove that fast transactions, as well as a weaker definition of fast transactions are impossible.

Our impossibility results are extremely useful because they clarify the limitations of real data stores and prevent practitioners' from chasing impossible designs.

1.5 Roadmap

The thesis consists of two parts. In the first part (chapters 2 and 3) we investigate general concepts regarding consensus and SMR. Specifically, in Chapter 2 we study the very definition and feasibility of a leaderless consensus algorithm. We provide a definition and present an eventually synchronous deterministic leaderless consensus algorithm. Among others, we use this consensus algorithm in an SMR implementation and show its robustness against arbitrary server failures. Then, in Chapter 3 we prove the surprising result that SMR is more expensive than a repetition of consensus instances. In the second part (Chapter 4), we prove that transactions cannot be fast in an asynchronous fault-tolerant system. Finally, in Chapter 5 we conclude this thesis and discuss future research directions.

Consensus and State Machine Replication

Part I

2 Leaderless Consensus

Unlike classic synchronous consensus algorithms, eventually synchronous consensus algorithms are leader-based. A leader helps them converge fast when the system is initially synchronous, but impacts performance when the system is not.

In this chapter, we ask whether, under eventual synchrony, it is possible to deterministically solve consensus without a leader. Actually, the fact that the weakest failure detector to solve consensus provides a leader [37] seems to indicate that some form of leader is inherently needed. We show that consensus algorithms need not be leader-based: there exists an eventually synchronous deterministic leaderless consensus algorithm.

In the rest of this chapter, we first address the question of the very meaning of a leaderless consensus algorithm. For pedagogical reasons, we do so in a shared memory system. Then, we present *Archipelago*, a new leaderless eventually synchronous consensus algorithm for shared memory and its message passing variant, and prove it correct. We also present a conjecture on how an Archipelago modification can tolerate Byzantine failures. Finally, we implement a leaderless state machine replication algorithm based on the message passing variant of Archipelago and illustrate its robustness in a failure scenario against ZooKeeper.

2.1 Introduction

Deterministic consensus algorithms that tolerate periods of asynchrony are considered robust because their safety is preserved even if the system is asynchronous [58]. These algorithms are typically *leader-based* [16, 35, 37, 72, 77]. A leader is convenient because it schedules commands when used in the context of state machine replication (SMR), helps ensure the uniqueness of a decision, and enforces a fast decision in good cases, i.e., when the system is synchronous and there is no failure or contention.

Unfortunately, the leader is a limitation in many ways, especially when used in the general context of SMR [6, 15, 30, 41, 57, 60, 64, 91, 96, 121, 125]. The leader is often considered the bottleneck in large distributed systems [64, 96]. A faulty leader has to be replaced through view-change procedures that are usually complex and error-prone [6]. In periods of asynchrony, a correct leader can be wrongly suspected. In addition, the choice of the timeout before suspecting a leader impacts the performance of practical systems [64, 100]. For example, the leader election of Zookeeper takes at least 200 ms [67] but a view change may delay concurrent operations by two orders of magnitude [57].

A lot of work has been devoted to minimizing the role of the leader by, for example, changing the leader frequently [30, 125] or tolerating multiple concurrent leaders [57, 91]. Note however that this work only eliminates the leader from the SMR algorithm. The underlying consensus algorithm for a single SMR slot or consensus instance remains Paxos-like and hence leader-based. In this work, we investigate whether an algorithm is leaderless firstly at the level of a single consensus instance. Hence, the leaderless property of the overlying SMR is obtained naturally by combining multiple consensus instances.

The natural question arises: Is it possible to eliminate the very notion of a leader completely in a deterministic consensus algorithm and still tolerate periods of asynchrony? Actually, the facts that (a) the weakest failure detector to solve consensus provides a leader [37] and that (b) one correct process must have as many eventually timely links as there can be failures [7, 26], in order to solve consensus in eventually synchronous systems, seem to indicate that some form of leader is inherently needed.

To address this question it is necessary to first define what a leaderless algorithm is. Although motivated by a message passing system, we start by doing so, for pedagogical reasons, in a shared memory system, before going to message passing. Intuitively, we capture the intuition that a leaderless consensus algorithm would be an algorithm that terminates and solves consensus even in scenarios where an adversary can suspend any process within each round of an eventually synchronous execution. In fact, the idea can be generalized to not only suspending 1 process but suspending k . We define the notion of a synchronous- k (which reads “synchronous minus k ”) round-based model where executions are (eventually) synchronous and at most $k < n$ processes can be suspended per round. A *leaderless* algorithm is one that decides in an eventually synchronous-1 (or \diamond synchronous-1) system.

With this definition of a leaderless algorithm at hand, we face the second challenge of asking whether we can devise a deterministic algorithm that solves consensus in an eventually synchronous-1 system. To illustrate the difficulty, assume the system is synchronous-1 from the start and consider the classic idea of exchanging values in rounds and adopting the maximum one. Because the adversary can suspend the process with the maximum value for as long as it wants, it is hard to guarantee that this process decides the same value as others.

Despite this powerful adversary, we show the existence of an eventually synchronous consensus algorithm that is leaderless. This algorithm, called *Archipelago*, builds upon a new variation of an adopt-commit object [52] whose invocation by different processes help them converge towards the same output value without a leader. Proving its leaderless property is not immediate because it requires at least $n \geq 3$ processes.

Based on this leaderless algorithm for shared memory, we derive algorithms for other models. First, we generalize it to the message passing model with crash failures. Second, we discuss how to transform this message passing leaderless algorithm to tolerate Byzantine failures. Finally, based on a sequence of Archipelago consensus instances, we build a state machine replication (SMR) algorithm, called *ArchSMR*. Our comparison against Apache ZooKeeper [64] in a distributed setting, shows the robustness of *ArchSMR* that keeps treating requests, while the ZooKeeper service gets disrupted.

Roadmap. Section 2.2 presents our model. Section 2.3 formalizes the notion of a leaderless consensus algorithm and discusses why well-known leader-based consensus algorithms do not fulfill this property. Section 2.4 presents a leaderless consensus algorithm in shared memory that makes use of a sequence of modified adopt-commit objects. Section 2.5 describes the proof, which we believe is interesting in its own right. Section 2.6 discusses the message passing leaderless algorithms that tolerate crash and Byzantine failures. Section 2.7 shows empirically the robustness advantage of an SMR based on Archipelago over ZooKeeper. Finally, Section 2.8 lists the related work.

2.2 Model

We consider an *asynchronous* shared-memory model with n processes $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$. Processes have access to (an infinite) set \mathcal{R} of atomic registers that can each store values from a set \mathcal{V} . Initially, all registers contain the initial value \perp . For notational simplicity, we assume that \mathcal{R} includes an infinite set of single-writer multi-reader (SWMR) arrays of n registers each. We denote these arrays as $\mathcal{R}_1, \mathcal{R}_2, \dots$ where a process p_i can write locations $\mathcal{R}_1[i], \mathcal{R}_2[i], \dots$. Processes *communicate* by reading from and writing to atomic registers. A *process* is a state machine that can change its state as a result of reading a register or writing to a register. An *algorithm* is the state machine for each process. A *configuration* corresponds to the state of all processes and the values in all registers in \mathcal{R} . An *initial configuration* is a configuration where all processes are in their initial state and all registers in \mathcal{R} contain value \perp .

When a process p performs a *read* or a *write*, we say that p performs a read or write *event* respectively. An *execution* corresponds to an alternating sequence of configurations and events, starting from an initial configuration. For example, in the execution

$\alpha = C, \text{read}(r, v)_p, C', \text{write}(r', v')_{p'}, C''$ we have processes $p, p' \in \mathcal{P}$, registers $r, r' \in \mathcal{R}$, values $v, v' \in \mathcal{V}$, and configurations C, C', C'' where C is an initial configuration, and the system moves from configuration C to C' when p reads v from r and from C' to C'' when p' writes v' to r' . We assume that all executions are *well-formed*, which roughly speaking means, that for a process p to perform an event after configuration C in an execution, there must be a transition specified by p 's state machine from p 's state in C . In this work, we consider deterministic algorithms and hence the initial state of processes and the sequence of processes that take steps uniquely defines a unique well-formed execution.

An execution α' is called an *extension* of a finite execution α if α is a prefix of α' . Two executions α and β are *equal* if both executions contain the exact same configurations and events in the same order. An *algorithm* \mathcal{A} is a set of state machines for each process and their initial states.

Synchronous- k execution. After setting up the main characteristics of our model, we can now define what it means for an execution to be synchronous in a shared-memory system.

Our definition is inspired by message passing, because our main motivation is to devise a leaderless consensus algorithm for the message passing model. In message passing models, during periods of synchrony behavior there is a bound on the time needed for a message to propagate from one process to another and for the receiver to process this message. Hence, in message passing we can divide time into rounds [48] such that, in each round, every process p does the following: (i) sends a message to every other process in the system, and (ii) delivers any message that was sent to p and performs some local computation.

We adapt this notion of synchrony to shared memory. We assume that processes take steps in rounds. Specifically, in each round, every process p_i does the following: (i) performs a write in some $\mathcal{R}_j[i]$, and (ii) collects all the values written in array \mathcal{R}_j . In one round, different processes can read from different arrays.

We precisely define this notion of synchrony below. A *collect* by a process p_i on an array \mathcal{R}_j is defined as a sequence of n read events: $\text{collect}(\mathcal{R}_j)_{p_i} = \text{read}(\mathcal{R}_j[1], \cdot)_{p_i}, \dots, \text{read}(\mathcal{R}_j[n], \cdot)_{p_i}$. By abuse of notation and because for defining collect we are not interested in the read values, we use the “ \cdot ” symbol. Similarly, in the rest of this chapter, we use \cdot at any point where we do not know or are not interested in the exact operation, value, etc. We define a *step* of \mathcal{R}_j by a process p_i as a write event and then a collect on \mathcal{R}_j . So, $\text{step}(\mathcal{R}_j)_{p_i} = \text{write}(\mathcal{R}_j[i], \cdot)_{p_i}, \text{collect}(\mathcal{R}_j)_{p_i}$. A *round* consists of all the write events $\text{write}(\mathcal{R}_{j_1}[1], \cdot)_{p_1}, \dots, \text{write}(\mathcal{R}_{j_n}[n], \cdot)_{p_n}$, followed by a sequence $\text{collect}(\mathcal{R}_{j_1})_{p_1}, \dots, \text{collect}(\mathcal{R}_{j_n})_{p_n}$ of collects by the exact same processes that performed a write event. Note that indices j_a and j_b could be the same for $a \neq b$. For example, if we only consider two processes $\{p_1, p_2\}$, then a round r could be the following sequence of events $r = \text{write}(\mathcal{R}_{j_1}[1], \cdot)_{p_1}, \text{write}(\mathcal{R}_{j_2}[2], \cdot)_{p_2}, \text{collect}(\mathcal{R}_{j_1})_{p_1}, \text{collect}(\mathcal{R}_{j_2})_{p_2}$.

	1	2	3	4	5	6	7	8	9	10	11
p_1	$step(\mathcal{R}_5)_{p_1}$	X	$step(\mathcal{R}_2)_{p_1}$	$step(\mathcal{R}_6)_{p_1}$	$step(\mathcal{R}_3)_{p_1}$	X	$step(\mathcal{R}_3)_{p_1}$	$step(\mathcal{R}_2)_{p_1}$	$step(\mathcal{R}_1)_{p_1}$	$step(\mathcal{R}_4)_{p_1}$	$step(\mathcal{R}_1)_{p_1}$
p_2	$step(\mathcal{R}_2)_{p_2}$	$step(\mathcal{R}_4)_{p_2}$	X	X	X	$step(\mathcal{R}_2)_{p_2}$	$step(\mathcal{R}_1)_{p_2}$	X	X	X	X

Figure 2.1 – Graphical depiction of a synchronous–1 execution.

To capture the notion that a process is *suspended* in a round r , we denote by $r|_{-\mathcal{P}_s}$ (notice the minus – in the notation) all the steps except the ones taken by processes in \mathcal{P}_s . For instance, for the above sequence r , we have $r|_{-\{p_1\}} = \text{write}(\mathcal{R}_{j_2}[1], \cdot)_{p_2}, \text{collect}(\mathcal{R}_{j_2})_{p_2}$.

We say that an execution is *synchronous– k* (which reads “synchronous minus k ”) if α is equal to a sequence of rounds $r_1|_{-\mathcal{P}_{s_1}}, r_2|_{-\mathcal{P}_{s_2}}, r_3|_{-\mathcal{P}_{s_3}}, \dots$ and $|\mathcal{P}_{s_i}| \leq k$ for $i \geq 1$. In other words, at most k processes can be *suspended* in each round. A suspended process p in a round r performs no events in r . For this reason, we call such an execution “synchrony minus k ,” since all processes except k behave synchronously in each round. We say that an infinite execution α is *eventually synchronous– k* (or \diamond synchronous– k) if an infinite suffix of α is equal to a synchronous– k execution. Naturally, a synchronous– k execution for $k = 0$ corresponds to a fully synchronous execution, while synchronous– k with $k > 0$ allows for some asynchrony in an execution.

In a synchronous– k or \diamond synchronous– k execution α , we say that a round r' occurs after round r if the events of round r' appear after the events of round r in α .

We say that a process p is *correct* in an infinite execution α if p is not suspended forever in α . More precisely, a process p is *correct* in an infinite execution if, for every round r there exists a later round r' such that process p is not suspended in r' . Figure 2.1 depicts a synchronous–1 execution for two processes p_1 and p_2 . Processes p_1 and p_2 take steps in a sequence of rounds, starting from round 1 and ending in round 11. The X symbol in a round indicates that the process is suspended in this round. In Figure 2.1, both processes perform steps in the first round, process p_1 in array \mathcal{R}_5 , while process p_2 in array \mathcal{R}_2 . Then, in the next round, process p_1 is suspended, etc.

Our shared-memory model is inspired by message-passing models [10, 105, 106, 108, 109] that allow for message omissions in each round, as well as by shared-memory models that combine an update and a collect (or scan), such as the iterated immediate snapshot model [24, 25]. Nevertheless, we opted for our simple model that mainly refers to synchronous executions and can easily be used to capture the notion of a leaderless algorithm. Additionally, note that in our model, a process is fully suspended (i.e., all communication from and to this process is omitted) in a round or not.

Consensus. In consensus [33], each process proposes a value by invoking a $\text{propose}(v)$ function and then all processes have to decide on a single value. Consensus is defined by the following three properties. *Validity* states that a value decided was previously proposed. *Agreement* states that no two processes decide different values, and *termination* states that every correct process eventually decides. We say that consensus algorithm *decides* in an execution α if a $\text{propose}(v)$ function call by some process p returns in α .

2.3 Leaderless Termination

We introduce a new liveness property that captures the notion that an algorithm can terminate without a leader.

Definition 1 (Leaderless Termination). *A consensus algorithm \mathcal{A} satisfies leaderless termination if, in every $\diamond\text{synchronous}-1$ execution of \mathcal{A} , every correct process decides.*

We can now define what it means for a consensus algorithm to be leaderless.

Definition 2 (Leaderless Consensus Algorithm). *A consensus algorithm is leaderless if it satisfies validity, agreement and termination, as well as leaderless termination.*

Note that in Definition 2, termination does not imply leaderless termination (see Section 2.5). A consensus algorithm that is not leaderless, is called *leader based*. To the best of our knowledge, this is the first formal definition of what leaderless means. The definition can be easily extended to the message-passing model, as we describe in Section 2.6.

Intuitively, Definition 2 is based on the idea that a leader-based algorithm encompasses the notion of a leader process. A process has first to become a leader and subsequently the leader process should be able to continuously perform a number of rounds in order for the consensus algorithm to decide. Therefore, after a process becomes a leader, the leader has to be unsuspended for at least one round in order for the consensus algorithm to decide. This implies that a potential adversary could prevent a leader-based consensus algorithm from deciding by selectively suspending a process the moment it becomes a leader. Therefore, an algorithm that decides even if we have the right to suspend one process per round, has to be a leaderless algorithm. Note that we do not argue that a leaderless algorithm always has an advantage over a leader-based algorithm, because it always decides in an $\diamond\text{synchronous}-1$ execution. For example, in an execution with no suspensions, a leader-based algorithm could perform better. The reason we devise Definition 2 is to capture whether an algorithm is leaderless or leader based.

Naturally, we do not argue that this is the only way to capture the notion of a leaderless algorithm. As a matter of fact, settling to Definition 2 was an arduous task. For instance,

one might be tempted to define a consensus algorithm as leaderless, if during an execution, irrespectively of which process crashes (i.e., gets suspended forever), the algorithm decides in the exact same number of rounds. More specifically, the idea would be to “stop” an execution in a bivalent state and examine what is the minimum number of rounds the algorithm needs to decide when crashing different processes. If the number of rounds is the same irrespectively of which process we crash, this suggests that no process has more responsibility than another and hence the algorithm is leaderless. However, we favor Definition 2, because an algorithm that satisfies our definition is robust against adaptive behavior of a dynamic adversary.

Examples. It is easy to design a consensus algorithm that decides in finite time in all synchronous-1 executions, but could however violate safety (e.g., agreement) in an \diamond synchronous-1 execution. For completeness, in what follows, we describe such an algorithm. We present Algorithm 1, an example of a consensus algorithm that decides in every synchronous-1, but that violates safety (i.e., agreement) when executed in an \diamond synchronous-1 execution.

Algorithm 1 Consensus algorithm that correctly decides in every synchronous-1 execution

```

1:  $\triangleright$  Shared state
2:  $Reg[n] \leftarrow \{\perp, \dots, \perp\}$   $\triangleright$  array of  $n$  single-writer multi-reader registers
3:
4:  $\triangleright$  process  $p_i$  proposes value  $v$ 
5: procedure propose( $v$ )
6:    $\triangleright$  first round
7:    $Reg[i] \leftarrow v$ 
8:    $vals \leftarrow \text{collect}(Reg) \setminus \{\perp\}$ 
9:   if  $\exists \langle \text{commit}, cv \rangle \in vals$  then
10:     $dv \leftarrow cv$   $\triangleright p_i$  was suspended in the first round, hence adopt committed value
11:  else
12:     $dv \leftarrow \max(\{v : v \in vals \vee \langle \cdot, v \rangle \in vals\})$ 
13:
14:   $\triangleright$  second round
15:   $Reg[i] \leftarrow \langle \text{commit}, dv \rangle$ 
16:  return  $dv$ 

```

Algorithm 1 satisfies validity, agreement, and decides in finite time in every synchronous-1 execution. Clearly, Algorithm 1 does not have a distinguished (leader) process that drives the decision, and the algorithm decides in two rounds if the system is synchronous-1. However, this algorithm is not leaderless according to Definition 1, because it does not tolerate asynchrony: in an \diamond synchronous-1, then the algorithm can violate safety.

We prove that Algorithm 1 satisfies validity, agreement, and decides in finite time in every

synchronous-1 execution below. *Validity.* Each process writes the proposed value in $Reg[i]$ (Line 7) and then collects (Line 8) all the values written in Reg . Hence, variable $vals$ contains only proposed values. Then, if there is a $\langle \text{commit}, cv \rangle$ pair in $vals$ the algorithm decides cv , stores $\langle \text{commit}, cv \rangle$ in $Reg[i]$ and returns (lines 9, 10, and 15). Otherwise, the algorithm retrieves the maximum value stored in $vals$, and hence retrieves a proposed value (Line 12). The process then stores $\langle \text{commit}, cv \rangle$ in $Reg[i]$ and returns (Line 15).

Agreement. Algorithm 1 satisfies agreement in a model with $n \geq 3$ processes. In a model with $n \geq 3$ processes, at least one process p performs steps in both rounds one and two. Process p writes $\langle \text{commit}, v \rangle$ (Line 15) in the second round and the algorithm decides v . If multiple processes were unsuspended in the first round, then all of the processes retrieve the same maximum value (Line 12), and hence write the exact same $\langle \text{commit}, dv \rangle$ pair in the second round (Line 15). Any process that was suspended in the first or second round, reads the committed value (Line 10) and hence decides on the same value.

In a model with $n = 2$ processes, Algorithm 1 could violate agreement, even in a synchronous-1 execution. For example, assume two processes p_1 and p_2 that propose v and v' respectively (with $v < v'$). Then, consider that process p_2 is suspended in the first round and process p_1 is suspended in the second round. Both processes p_1 and p_2 are unsuspended in the third round. In such an execution, p_1 writes v to $Reg[0]$ and then retrieves the maximum value in Reg , which is v . Then, in the second round, process p_2 writes v' to $Reg[1]$ and retrieves the maximum value in Reg , which now is v' . Hence in the third round, processes p_1 and p_2 decide v and v' respectively.

Algorithm 1 is a straight-forward algorithm. The real challenge, is to devise a leaderless consensus algorithm that decides in finite time in every \diamond synchronous-1 execution and never violates safety. We present such an algorithm in Section 2.4.

Consider Algorithm 2, a leader-based shared-memory algorithm that is at the heart of the well-known Paxos [77] algorithm. As a matter of fact, Algorithm 2 combined with a leader election algorithm corresponds to Paxos in shared memory.

In Algorithm 2, all processes share an array R of n single-writer multi-reader (SWMR) registers (Line 2); each element of R is a pair $\langle a, b \rangle$ where a corresponds to a value and b to a timestamp. Additionally, each process stores locally a ts value (Line 5). When a process p_i invokes $\text{propose}(v)$, p_i first stores its current timestamp value to $R[i]$ (Line 10). Then, p_i retrieves a value from array R , the one associated with the highest timestamp (Line 11). If no such value exists and hence $val = \perp$, then p_i proposes its own value v (Line 13). Otherwise, process p_i proposes the values that p retrieved from array R . Then, p_i stores the pair $\langle val, ts \rangle$ to array $R[i]$ (Line 14) and examines whether the highest timestamp in R is the one that p_i wrote (Line 15). If this is the case, the algorithm decides (Line 16), otherwise p_i increases ts and repeats the

Algorithm 2 Leader-based consensus algorithm

```

1: ▷ Shared state
2:  $R[n] \leftarrow \{\langle \perp, 0 \rangle, \dots, \langle \perp, 0 \rangle\}$  ▷ one SWMR registers per process
3:
4: ▷ Local state
5:  $ts \leftarrow i$  ▷ for process  $p_i$ 
6:
7: ▷ process  $p_i$  proposes value  $v$ 
8: procedure propose( $v$ )
9:   while true do
10:     $R[i].ts \leftarrow ts$ 
11:     $val \leftarrow \text{getHighestTspValue}(R)$ 
12:    if  $val = \perp$  then
13:       $val \leftarrow v$ 
14:     $R[i] \leftarrow \langle val, ts \rangle$ 
15:    if  $ts = \text{getHighestTsp}(R)$  then
16:      return  $val$ 
17:     $ts \leftarrow ts + n$ 

```

loop (Line 17).

We can think of Algorithm 2 using Paxos terminology [78]. A timestamp is used like the proposal (or ballot) number of Paxos. Getting the timestamp and retrieving the value with the highest associated timestamp constitutes the *prepare* phase. Then, writing the $\langle val, ts \rangle$ pair and checking whether the highest timestamp has not been modified constitutes the *propose* phase.

According to Definition 2, Algorithm 2 is leader based. In a \diamond synchronous-1 execution, where in each round, the adversary suspends the process who has written the highest value in the R array so far, then the algorithm never decides. Specifically, when a process p is about to check whether its timestamp ts is the highest timestamp (Line 15), an adversary can suspend p until some other process p' stores a timestamp ts' that is greater than ts in array R (Line 10). This can occur even if the execution is synchronous-1.

2.4 Archipelago: Leaderless Consensus

We introduce *Archipelago*, our new leaderless consensus algorithm. Archipelago is leaderless according to Definition 1, when $n \geq 3$ and never violates safety (i.e., validity and agreement). Let us first recall the adopt-commit object and present our adopt-commit-max variant used in Archipelago.

Adopt-commit-max implementation. The *adopt-commit object* [52] has the following specification. Every process p proposes an input value v to such an object and obtains an output, which consists of a pair $\langle d, v \rangle$; d can be either commit or adopt. The following properties are satisfied:

- **CA-Validity:** If a process obtains output $\langle \text{commit}, v \rangle$ or $\langle \text{adopt}, v \rangle$, then v was proposed by some process.
- **CA-Agreement:** If a process p outputs $\langle \text{commit}, v \rangle$ and a process q outputs $\langle \text{commit}, v' \rangle$ or $\langle \text{adopt}, v' \rangle$, then $v = v'$.
- **CA-Commitment:** If every process proposes the same value, then no process may output $\langle \text{adopt}, v \rangle$ for any value v .
- **CA-Termination:** Every correct process eventually obtains an output.

Algorithm 3 depicts a new implementation of an adopt-commit object. It differs from the classic implementation [52] in that if the collect of A by process p that proposes v returns different values, then p stores $\langle \text{adopt}, mv \rangle$ to array B (Line 8) instead of storing $\langle \text{adopt}, v \rangle$, where mv is the maximum of the values collected from A ($\max(S_A)$). Additionally, if all pairs collected from B are of the form $\langle \text{adopt}, \cdot \rangle$, then process p returns $\langle \text{adopt}, mv \rangle$, where mv is $\max(S_A)$ (Line 12). Note that Algorithm 3 is just a different implementation of the classic implementation [52] and that the main properties of an adopt-commit object remain the same. These modifications are crucial in order for Archipelago to satisfy leaderless termination.

Algorithm 3 The adopt-commit-max algorithm

```

1:  $\triangleright$  Shared state
2:  $A$  and  $B$ , two arrays of  $n$  single-writer multi-reader registers, all initially  $\perp$ 
3:
4: procedure propose( $v$ )  $\triangleright$  taken by a process  $p_i$ 
5:    $A[i] \leftarrow v$   $\triangleright$  step  $A$  starts
6:    $S_A \leftarrow \text{collect}(A)$   $\triangleright$  step  $A$  ends
7:   if ( $S_A \setminus \{\perp\} = \{v'\}$ ) then  $B[i] \leftarrow \langle \text{commit}, v' \rangle$   $\triangleright$  step  $B$  starts
8:   else  $B[i] \leftarrow \langle \text{adopt}, \max(S_A) \rangle$   $\triangleright$  or step  $B$  starts
9:    $S_B \leftarrow \text{collect}(B)$   $\triangleright$  step  $B$  ends
10:  if  $S_B \setminus \{\perp\} = \{\langle \text{commit}, v' \rangle\}$  then return  $\langle \text{commit}, v' \rangle$ 
11:  else if  $\langle \text{commit}, v' \rangle \in S_B$  then return  $\langle \text{adopt}, v' \rangle$ 
12:  else return  $\langle \text{adopt}, \max(S_A) \rangle$ 

```

In what follows we prove the correctness of Algorithm 3, which is similar to that of an adopt-commit object [52]. Algorithm 3 satisfies CA-Validity (the max function preserves validity) and

CA-Termination (Algorithm 3 does not use waiting or loops). To prove CA-Agreement and CA-Commitment, we first prove the following lemma.

Lemma 1. *If B contains two entries (commit, v_1) and (commit, v_2) , then $v_1 = v_2$.*

Proof. Assume not. Since every process writes in A and B at most once, it must be that some process p_1 wrote (commit, v_1) and some other process p_2 wrote (commit, v_2) . Thus, it must be that p_1 wrote v_1 in A , took a collect of A and only saw v_1 in that collect. Similarly, it must be that p_2 wrote v_2 in A , took a collect of A and only saw v_2 in that collect. This is impossible: since the processes update A before collecting, it must be that either p_1 saw p_2 's value, or vice-versa. We have reached a contradiction. \square

CA-Agreement. In order for a process p to commit v , p must write v to A , collect A and see only entries equal to v ; p must then write $\langle \text{commit}, v \rangle$ to B , collect B and see only entries equal to $\langle \text{commit}, v \rangle$ and finally return $\langle \text{commit}, v \rangle$.

Assume by contradiction that process p commits v and some process q commits or adopts $v' \neq v$. q 's collect of B cannot include the $\langle \text{commit}, v \rangle$ entry written by p , otherwise q would adopt v (remember that by Lemma 1, q cannot see any entry $\langle \text{commit}, v' \rangle$ with $v' \neq v$ in B since p writes $\langle \text{commit}, v \rangle$ to B). Therefore, q 's collect of B must happen before p 's write to B . Furthermore, q 's collect of B must include some entry $e = \langle \cdot, v' \rangle$ with $v' \neq v$ (written either by q or some other process). But then p 's collect of B (which is after p 's write to B and therefore after q 's collect of B) will also include e , and thus p cannot commit v . We have reached a contradiction.

CA-Commitment. Assume all proposed values are equal. Then no process can write $\langle \text{adopt}, \cdot \rangle$ in B ; B contains only entries of the form $\langle \text{commit}, \cdot \rangle$. By Lemma 1, all such entries have equal values, so all processes that return must commit.

Archipelago Algorithm. The main idea of Algorithm 4 is to use a sequence of adopt-commit-max objects,¹ in combination with a max register. A *max register* r is a wait-free register that provides a write operation, as well as a read operation (`readmax`) that retrieves back the largest value that was previously written to r [12]. A max register can be implemented by simply letting each process write to a single-writer multi-reader register and then collecting all the values written by all processes and taking the maximum. The adopt-commit-max objects ensure safety, while the max register and the maximum component (Line 12) of the adopt-commit-max help the processes converge on the value the processes propose to some later

¹For brevity, we use the terms “adopt-commit-max implementation” and “adopt-commit-max object” interchangeably.

adopt-commit-max object in the sequence of adopt-commit-max objects. In a synchronous-1 execution, the processes converge on one value and there is an adopt-commit-max object where all processes propose this exact single value. Then, due to CA-commitment property of the adopt-commit-max object, the adopt-commit-max outputs $\langle \text{commit}, \cdot \rangle$ and Archipelago decides in finite time.

Algorithm 4 contains Archipelago. In Archipelago, all processes share an infinite sequence of adopt-commit-max objects (C) and a max register m (lines 14 to 16). Additionally, each process stores locally in variable c the index of the next adopt-commit-max object to be used (Line 19), thus a process intends to use $C[c]$. Each process p in Archipelago first writes $\langle c, v \rangle$ to m (Line 23) and then retrieves the maximum tuple $\langle c', v' \rangle$ stored in m (Line 24). Note that values c and v are not necessarily equal to c' and v' . Process p then proposes value v' to $C[c']$ (i.e., p invokes $C[c'].\text{propose}(v')$ at Line 25) and sets c to correspond to the next to-be-used adopt-commit-max object (Line 26). If process p receives a commit response from some adopt-commit-max object (Line 28), then process p decides and can return. Otherwise, when process p receives an $\langle \text{adopt}, v'' \rangle$ response, process p stores this result in the m register (Line 27) and repeats the loop.

Algorithm 4 The Archipelago leaderless consensus algorithm

```

13: ▷ Shared state
14:  $C[0, \dots, +\infty]$ , an infinite array of adopt-commit-max objects in their initial state
15:  $m$ , a max register object that initially contains  $\langle 0, \perp \rangle$ .
16: Note that  $\langle x, y \rangle > \langle x', y' \rangle$  if  $x > x'$  or  $(x = x' \text{ and } y > y')$ 
17:
18: ▷ Local state
19:  $c$ , index of the adopt-commit-max object to use, initially 0
20:
21: procedure propose( $v$ )
22:   while true do
23:      $m.\text{write}(\langle c, v \rangle)$  ▷ step  $R$  starts
24:      $\langle c', v' \rangle \leftarrow m.\text{readmax}()$  ▷ step  $R$  ends
25:      $\langle \text{control}, v'' \rangle \leftarrow C[c'].\text{propose}(v')$ 
26:      $c \leftarrow c' + 1$ 
27:     if  $\text{control} = \text{adopt}$  then  $v \leftarrow v''$ 
28:     else return  $v''$ 

```

Recall from Section 2.2 that a step consists of a write and a subsequent collect. Algorithm 4 operates by repeatedly performing three steps. First, a process performs what we call an R step (lines 23-24). Namely, an R step corresponds to the writing and the reading of max register m . Then, during the proposal of an adopt-commit-max object (Line 25), a step A is performed (lines 5-6) in the underlying adopt-commit-max object (Algorithm 3). Finally, a step B starts either in Line 7 or in Line 8 of Algorithm 3 and the subsequent collect takes place in Line 9. We

use the notion of steps R , A , and B in the next section, where we prove that Archipelago is a leaderless consensus algorithm.

The cautious reader might think that by solving consensus in an \diamond synchronous-1 execution with Archipelago, we could implement the Ω failure detector [37]. We could then augment Algorithm 2 with Ω so that Algorithm 2 decides in every \diamond synchronous-1 execution. There is work that implements Ω in crash-recovery settings, however only when a crashed process can recover a finite number of times [33, 50, 93]. This is in contrast to our model, where a process can crash and recover an infinite number of times. In other words, in our model every process is *unstable* [93] and hence questions the existence of Ω in our model.

2.5 Archipelago: Proof of Correctness

Archipelago is a leaderless consensus algorithm. First we show that it satisfies the consensus properties (validity, agreement, and termination under \diamond synchrony) and then we prove that it provides leaderless termination, which is more interesting and significantly more challenging. Note that Archipelago solves multi-valued consensus. Naturally, we could have presented and proved correct a modified version of Archipelago for binary consensus. However, we do not believe that such an approach would simplify either the presentation or the proof of Archipelago as we explain later on.

Validity, agreement, termination. Algorithm Archipelago satisfies *validity*. We prove that if an adopt-commit-max object $C[c]$ returns a $\langle \cdot, v \rangle$ tuple, then v was proposed by some process. We can easily show this using induction. For $c = 0$, this is clearly the case, since all the values that were proposed to $C[0]$ are written in m and were initially proposed. Let $c \geq 0$. Assume that for every adopt-commit-max object $C[c']$ with $c' \leq c$, $C[c']$ returns a value that was initially proposed by some process. Then, for a value v to be proposed to $C[c + 1]$, this means that a process read $\langle c + 1, v \rangle$ from m (Line 24). This implies that at some point, some process p writes $\langle c + 1, v \rangle$ to m (Line 23). But for this to happen, p retrieved $\langle \text{adopt}, v \rangle$ from an adopt-commit-max object $C[c']$ with $c' < c + 1$ and by induction, this means that v is a proposed value. Since all the values returned by adopt-commit-max objects are proposed, and Archipelago decides (Line 28) upon a value that Archipelago retrieves from some adopt-commit-max object, Archipelago satisfies validity.

Algorithm Archipelago satisfies *agreement*. To see this, assume by way of contradiction that two processes p and p' decide on different values v and v' respectively. This means that process p returned v after receiving a $\langle \text{commit}, v \rangle$ response for an adopt-commit-max object $C[c]$ and process p' received a $\langle \text{commit}, v' \rangle$ response for an adopt-commit-max object $C[c']$. Because the adopt-commit-max object satisfies CA-agreement, it has to be the case that $c \neq c'$,

otherwise $v = v'$. Without loss of generality, assume that $c < c'$. All the processes (including p') that received a response from $C[c]$ either received $\langle \text{commit}, v \rangle$ or $\langle \text{adopt}, v \rangle$ due to the agreement property of the adopt-commit-max object. Hence, all processes that write to m (Line 23), write $\langle c + 1, v \rangle$, since they retrieved v from $C[c]$. Therefore, all possible values that are proposed to the $C[c + 1]$ adopt-commit-max object, propose v , and hence $C[c + 1]$ returns $\langle \text{commit}, v \rangle$. Similarly, all upcoming adopt-commit-max-objects return $\langle \text{commit}, v \rangle$ contradicting the fact that $C[c']$ ($c < c'$) responds with $\langle \text{commit}, v' \rangle$ with $v' \neq v$.

Leaderless termination. It is far from obvious that Archipelago satisfies leaderless termination. As a matter of fact, Archipelago does not provide leaderless termination for $n = 2$ processes. However, Archipelago satisfies leaderless termination for $n \geq 3$ processes. Before we describe the proof, we introduce some auxiliary notation.

Notation. For an execution α we say that a process p takes a step $A_i(v)$ when p performs an A step that belongs to adopt-commit-max object $C[i]$ (lines 5 and 6). We denote with $A_i^0(v)$ the fact that p is the first process that performed the A step for adopt-commit-max object $C[i]$ in execution α . Note that a single round might contain multiple $A_i^0(v)$ steps taken by different processes. We denote with $A_i^+(v)$ the fact that this step is not the first A step on $C[i]$. We denote with $B_i(\mathbf{1}, v)$ the B step of a process on adopt-commit-max object $C[i]$ that writes $\langle \text{commit}, v \rangle$ (lines 7 and 9). With $B_i(\mathbf{0}, v)$, we denote the B step of a process on adopt-commit-max object $C[i]$ that writes $\langle \text{adopt}, v \rangle$ (lines 8 and 9). Similarly to the notation of an A step, we use the notation $B_i^0(\mathbf{1}, v)$, and $B_i^+(\mathbf{1}, v)$. We say that in an execution α values v_1, v_2, \dots, v_k are proposed to $C[i]$ if there are steps $A_i(v_j) \forall 1 \leq j \leq k$ in α . We denote with $R\langle c, v \rangle$ the R step of a process and the fact that the process read $\langle c, v \rangle$ as the maximum value in m (lines 23 and 24). As with steps A and B , we use the $R^0\langle c, v \rangle$ and $R^+\langle c, v \rangle$ notation. Specifically, with $R^0\langle i, \cdot \rangle$ we denote the first R step that reads $\langle i, \cdot \rangle$. Note that in this notation when we have $A_i(v)$ and $B_i(\cdot, v)$, this v is the value that is written, while in $R\langle c, v \rangle$ the value v is read from m . Furthermore, note that R is not part of an adopt-commit-max operation like the A and B steps and hence has no subscript.

$n = 2$ processes. For $n = 2$ processes, we can devise a synchronous-1 execution in which the Archipelago algorithm never decides. This execution is depicted in Figure 2.2. Figure 2.2 has a pattern that repeats every 5 rounds (light-green boxes). In Figure 2.2, processes p_1 and p_2 propose values v' and v respectively with $v' > v$. In the first round, process p_1 is suspended, so process p_2 performs an R step, writes $\langle 0, v \rangle$, and retrieves $\langle 0, v \rangle$ from m . Then, in the second round both processes p_1 and p_2 take steps. Process p_1 writes $\langle 0, v' \rangle$ and retrieves $\langle 0, v' \rangle$ since $\langle 0, v' \rangle > \langle 0, v \rangle$. In the same round, p_2 writes v to $C[0].A[2]$. Then, in the third round, when process p_1 takes an A step it writes value v' in $C[0].A[1]$ and when p_1 collects the values written in array A (Line 6), p_1 sees that there are two different values (v and v') in $C[0].A$. Therefore, in

2.5. Archipelago: Proof of Correctness

p_1	X	$R^+\langle 0, v' \rangle$	$A_0^+(v')$	$B_0^0(\mathbf{0}, v')$	X	X	$R^+\langle 1, v' \rangle$	$A_1^+(v')$	$B_1^0(\mathbf{0}, v')$	X	X	...
p_2	$R^0\langle 0, v \rangle$	$A_0^0(v)$	X	X	$B_0^+(\mathbf{1}, v)$	$R^0\langle 1, v \rangle$	$A_1^0(v)$	X	X	$B_1^+(\mathbf{1}, v)$	$R^0\langle 2, v \rangle$	

Figure 2.2 – With 2 processes, Archipelago might never decide in a synchronous–1 execution ($v' > v$).

the fourth round, when process p_1 performs a B step, it retrieves back $\langle \text{adopt}, v' \rangle$. Process p_2 takes a B step in the fifth round after being suspended in the third and fourth rounds, p_2 writes $\langle \text{commit}, v \rangle$ in $C[0].B[2]$, and then during the collect of B , p_2 sees that $\langle \text{adopt}, v' \rangle$ is written in $C[0].B[1]$ and p_2 returns $\langle \text{commit}, v \rangle$ (Line 11). Afterwards, starting from the sixth round the processes behave in the exact same way: processes p_1 and p_2 propose v' and v to the next adopt-commit-max object respectively. This can happen ad infinitum and Archipelago never decides.

$n \geq 3$ processes. We consider synchronous–1 executions that start from an arbitrary, albeit valid (i.e., state corresponds to a configuration in a well-formed execution), initial state. We prove that in every synchronous–1 execution, irrespectively of the initial state, Archipelago terminates in finite time. Therefore, in every \diamond synchronous–1 execution, eventually the execution becomes synchronous–1 and hence Archipelago decides in finite time.

Theorem 1. *Archipelago satisfies leaderless termination for $n \geq 3$.*

To prove Theorem 1, we show that as Archipelago traverses adopt-commit-max objects, the current minimal value, among those values still being proposed to adopt-commit-max objects, eventually gets eliminated (i.e., processes only propose larger values in later adopt-commit-max objects). Therefore, eventually only one value gets proposed to some adopt-commit-max object, and every correct process decides.

Note that to prove Theorem 1, we have to show that Archipelago terminates in finite time in every synchronous–1 execution, irrespectively of the initial state (i.e., any state that corresponds to a configuration in a well-formed execution). Therefore, in every \diamond synchronous–1 execution, eventually the execution becomes synchronous–1 and hence Archipelago decides in finite time.

Before we prove Theorem 1, we first need to prove some auxiliary lemmas.

Lemma 2. *If an execution α contains step $R^0\langle i, v \rangle$, then for any step $R\langle j, v' \rangle$ with $j > i$ that is in α , it is the case that $v' \geq v$.*

Proof. Consider an execution α that contains a step $R^0\langle i, v \rangle$ in a round r taken by process

p . Then, when process p continues, p proposes value v to adopt-commit-max object $C[i]$. Similarly and since each process retrieves the maximum value when reading array R (Line 24), any later process that performs an R step in round r or after r reads at least $\langle i, v \rangle$, and hence retrieves a value at least as great as v . Note that a process that performs an R step in round r cannot read $\langle j, v' \rangle$ with $j > i$ and $v' < v$, since process p takes step $R^0 \langle i, v \rangle$. Hence, all values that are proposed to adopt-commit-max object $C[j]$ ($j \geq i$) are $\geq v$ and therefore for any step $R \langle j, v \rangle$ with $j > i$, it holds that $v' \geq v$. \square

Lemma 3. *If an execution α contains step $B_i^0(\mathbf{1}, v)$, then Archipelago decides v in α .*

Proof. Assume an execution α contains step $B_i^0(\mathbf{1}, v)$ in round r . If a process p takes a step $B_i(\cdot, \cdot)$, then p definitely takes the step in a round k with $k \geq r$. Therefore, process p sees $\langle \text{commit}, v \rangle$ when collecting B (Line 9) and either returns $\langle \text{commit}, v \rangle$ (Line 10 and then Line 28) and decides, or returns $\langle \text{adopt}, v \rangle$ (Line 11). Due to CA-agreement, p cannot return $\langle \text{commit}, v' \rangle$ $\langle \text{adopt}, v' \rangle$ with $v' \neq v$. Thus, process p proposes v in adopt-commit-max object $C[i + 1]$. However, when all processes propose the same value v to adopt-commit-max object $C[i + 1]$, then Archipelago decides v . \square

Lemma 4. *If an execution α contains at least two steps $A_i^0(v)$ from processes p and p' ($p \neq p'$), and there is no process performing step $A_i^0(v')$ with $v' \neq v$ in α , then either p , or p' , or both perform step $B_i^0(\mathbf{1}, v)$ in α .*

Proof. Suppose that a round r contains two $A_i^0(v)$ events by processes p and p' respectively. Since in a round, there can be at most one suspended process, this means that at least one of the processes p and p' take a step in round $r + 1$. Since both processes p and p' write value v in array $C[i].A$, and no process wrote another value in $C[i].A$ during that round, v is the only value that p and p' read when collecting A , and hence in the upcoming step in round $r + 1$, at least one of the two processes writes $B_i^0(\mathbf{1}, v)$. \square

Roughly speaking, the following lemma states that if an execution contains a step $A_i^0(v')$ where $v' > \min(\{v : \exists A_i(v) \in \alpha\})$, then any value proposed to a later adopt-commit-max object (i.e., written in A) is greater than $\min(\{v : \exists A_i(v) \in \alpha\})$, namely is greater than the minimum value proposed in adopt-commit-max object $C[i]$.

Lemma 5. *In an execution α , consider $\mathcal{V}_f = \{v : \exists A_i(v) \in \alpha\}$ and let v_m be $\min(\mathcal{V}_f)$. If there is a step $A_i^0(v) \in \alpha$ with $v > v_m$, then for any step $A_j(v') \in \alpha$ with $j > i$, it is the case that $v' > v_m$.*

Proof. Because execution α contains step $A_i^0(v)$ with $v > \min(\mathcal{V}_f)$, any step A_j with $j > i$ on adopt-commit-max object $C[j]$ sees value v written in array A (Line 8) and hence adopts a value v' with $v' \geq v > v_m$.

2.5. Archipelago: Proof of Correctness

	...	$r-1$	r	$r+1$...	$r+x-1$	$r+x$	$r+x+1$	$r+x+2$	$r+x+3$...
p_a		.	$A_i^0(v_m)$	X	X	X	X	$B_i^+(1, v_m)$	$R(i+1, v_m)$.	
p_b		$A_i^+(v)$	$B_i^0(0, v)$	X	X	.	
p_c		
\vdots											

\leftarrow $\nexists R(i+1, \cdot)$ step before round $r+x+2$.

Figure 2.3 – Execution pattern that appears when the minimum value propagates to the next adopt-commit-max object ($x \geq 2$).

□

To prove Theorem 1 we show that as Archipelago traverses adopt-commit-max objects, the current minimal value, among those values still being proposed to adopt-commit-max objects, eventually gets eliminated (i.e., processes only propose larger values in later adopt-commit-max objects). Specifically, we show that in at most three consecutive adopt-commit-max objects, the minimal value gets eliminated. Since we have n processes, we can have at most n distinct proposed values. Therefore, using at most $3n$ adopt-commit-max objects, Archipelago decides in finite time. From the moment of synchrony, Archipelago needs $\mathcal{O}(n)$ rounds to decide.

Towards this goal, the following lemma is useful. Lemma 6 captures the idea that if in an execution α , the minimum value proposed to an adopt-commit-max object $C[i]$ appears in a later adopt-commit-max object $C[j]$ with $j > i$, then α contains a specific execution pattern. By execution pattern we mean, that some process has to take a step, then be suspended, then another process has to take some step, etc.

Figure 2.3 captures the fact that there is some process p_a that takes an $A_i^0(v_m)$ step and before p_a performs $B_i(1, v_m)$ some other process p_b performs $A_i^+(v)$ and $B_i^0(0, v)$, etc.

Lemma 6. *In an execution α , consider $\mathcal{V}_f = \{v : \exists A_i(v) \in \alpha\}$ and let v_m be $\min(\mathcal{V}_f)$. If Archipelago does not decide in α and there is a step $A_j(v_m) \in \alpha$ with $j > i$, then $\exists x \geq 2$ and $\exists p_a, p_b \in \mathcal{P}$ and round r such that p_a, p_b perform steps as depicted in Figure 2.3 and there is no $R(i+1, \cdot)$ step taken before round $r+x+2$.*

Proof. Suppose that α has no step $A_i^0(v_m)$ and hence α contains a step $A_i^0(v)$ with $v > v_m$. Then, due to Lemma 5, we know that for every $A_j(v')$ with $j > i$ it is the case that $v' > v_m$. But this implies that there is no $A_j(v_m)$ with $j > i$ in α and this is not the case we consider. Therefore, for an $A_j(v_m)$ to exist in α , execution α must contain $A_i^0(v_m)$.

Assume that process p_a takes step $A_i^0(v_m)$ in some round r . Lemmas 3 and 4 imply that if there is another $A_i^0(v_m)$ step in α taken by some process $p \neq p_a$, then the algorithm decides. Since in the lemma we assume that Archipelago does not decide, we can exclude this case and consider that there is at most one $A_i^0(v_m)$ in round r .

Suppose that process p_a takes a step in round $r + 1$. Then, process p_a takes a $B_i^0(\mathbf{1}, v_m)$ step since p_a was the process that first performed an A step on adopt-commit-max object $C[i]$. However, if process p_a takes a $B_i^0(\mathbf{1}, v_m)$, due to Lemma 3, the algorithm decides. Again, we do not consider this case. Similarly, if process p_a takes a B step in round $r + 2$, then process p_a takes a $B_i^0(\mathbf{1}, v_m)$ step and due to Lemma 3, the algorithm decides. Therefore, we need to consider the case where process p_a is suspended in both rounds $r + 1$ and $r + 2$. Process p_a can potentially be suspended for more rounds, up to round $r + x$ where $x \geq 2$. Therefore, for v_m to appear in a later adopt-commit-max object $C[j]$ with $j > i$ with an $A_j(v_m)$ step, execution α has to be similar to the execution depicted below.

	...	$r-1$	r	$r+1$...	$r+x-1$	$r+x$	$r+x+1$	$r+x+2$	$r+x+3$...
p_a		.	$A_i^0(v_m)$	X	X	X	X	$B_i(\mathbf{1}, v_m)$.	.	
p_b		
p_c		
\vdots											

We now show that there cannot be an $R(i+1, \cdot)$ step before round $r + x + 2$. Assume by way of contradiction that there exists an $R(i+1, \cdot)$ step before round $r + x + 2$ in α . If multiple such steps exist in α , consider the one that takes place in the earliest round. Suppose that this $R^0(i+1, v)$ has $v > v_m$. This means that a later process reads value $v > v_m$ and hence when later processes perform an R in some later round, they see a value (Line 24) greater than v_m and hence propose only values greater than v_m to upcoming adopt-commit-max objects (Lemma 2). This contradicts the fact that there is a $j > i$ with $A_j(v_m)$.

This means that if an $R^0(i+1, v')$ step appears before round $r + x + 2$ in α , then it has to be that $v' = v_m$. Suppose that this $R^0(i+1, v_m)$ is taken by some process p in round $r + y$. Before round $r + y$ process p has to take steps A_i and B_i since p performs the first $R^0(i+1, v_m)$ step. This means that value y has to be greater than 2, since otherwise it implies that step A_i taken by p occurs in a round smaller or equal than r . However, process p_a is the only process that takes an $A_i^0(v_m)$ in round r .

Since $R^0(i+1, v_m)$ occurs in round $r + y$, where $2 < y < x + 2$, then p must perform an $A_i(v)$ step in round $r + y - 2$ and a $B_i^0(\cdot, \cdot)$ step in round $r + y - 1$ (p cannot be suspended between

2.5. Archipelago: Proof of Correctness

$r + y - 2$ and $r + y$ because p_a is already suspended). If $v = v_m$, then p 's B_i step will be $B_i^0(\mathbf{1}, v_m)$ and so, due to Lemma 3, the algorithm decides (Line 10 and Line 28), which we assume does not happen in α . If $v > v_m$, then p 's B_i step will be $B_i^0(\mathbf{0}, v)$, which contradicts the fact that p does $R^0\langle i + 1, v_m \rangle$ immediately afterwards.

Therefore, there cannot be an $R\langle i + 1, \cdot \rangle$ step before round $r + x + 2$. This is depicted in the figure below where all rounds less than $r + x + 2$ highlighted in light-red cannot contain an $R\langle i + 1, \cdot \rangle$ step.

	...	$r-1$	r	$r+1$...	$r+x-1$	$r+x$	$r+x+1$	$r+x+2$	$r+x+3$...
p_a		.	$A_i^0(v_m)$	X	X	X	X	$B_i(\mathbf{1}, v_m)$.	.	
p_b		
p_c		
\vdots											

← $\nexists R\langle i + 1, \cdot \rangle$ step before round $r + x + 2$.

If between rounds r and $r + x + 1$ no other process performs a $B_i^0(\cdot, \cdot)$ step, then process p_a is the first to take a B-Step in adopt-commit-max object $C[i]$ and thus its B-Step is $B_i^0(\mathbf{1}, v_m)$. Hence Archipelago decides due to Lemma 3, which contradicts our initial assumption. Therefore, there is at least one process p_b that performs $B_i^0(\cdot, \cdot)$ between rounds $r + 1$ and $r + x + 1$. If process p_b takes step $B_i^0(\cdot, \cdot)$ in a round smaller than $r + x$, then it performs $R\langle i + 1, \cdot \rangle$ before round $r + x + 2$ since process p_b has to take continuous steps because p_a is suspended from round $r + 1$ to round $r + x + 1$, a contradiction. Therefore, process p_b performs a step $A_i(v)$ with $v > v_m$ in round $r + x - 1$ and $B_i^0(\mathbf{0}, v)$ in round $r + x$. The current execution is depicted below.

	...	$r-1$	r	$r+1$...	$r+x-1$	$r+x$	$r+x+1$	$r+x+2$	$r+x+3$...
p_a		.	$A_i^0(v_m)$	X	X	X	X	$B_i(\mathbf{1}, v_m)$.	.	
p_b		$A_i^+(v)$	$B_i^0(\mathbf{0}, v)$.	.	.	
p_c		
\vdots											

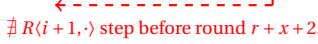
← $\nexists R\langle i + 1, \cdot \rangle$ step before round $r + x + 2$.

Due to Lemma 2, process p_b must be suspended in round $r + x + 1$, as well as in round $r + x + 2$. Since otherwise, if process p_b is not suspended in rounds $r + x + 1$ and $r + x + 2$, this implies

that process p_b takes an $R^0\langle i+1, v \rangle$ step, where $v > v_m$. Due to Lemma 2, this implies that no process proposes v_m to all upcoming adopt-commit-max objects, because all $R\langle i+1, \cdot \rangle$ appear after round $r+x+1$, which contradicts the if-statement of our lemma. Since process p_b is suspended in round $r+x+2$ and at most one process can be suspended in each round, process p_a takes an $R^0\langle i+1, v_m \rangle$ step in round $r+x+2$.

We are therefore in the following setting that is the exactly the same execution pattern as the one in Figure 2.3.

	...	$r-1$	r	$r+1$...	$r+x-1$	$r+x$	$r+x+1$	$r+x+2$	$r+x+3$...
p_a		.	$A_i^0(v_m)$	X	X	X	X	$B_i(\mathbf{1}, v_m)$	$R^0\langle i+1, v_m \rangle$.	
p_b		$A_i^+(v)$	$B_i^0(\mathbf{0}, v)$	X	X	.	
p_c		
\vdots											



To conclude, given an adopt-commit-max object $C[i]$ where the minimum value proposed is v_m , for value v_m to be proposed in the next adopt-commit-max object $C[i+1]$, it has to be that the execution is as shown in Figure 2.3. In other words, there is some process p_a that takes an $A_i^0(v_m)$ step alone and, before p_a performs $B_i(\mathbf{1}, v_m)$, some other process p_b performs $A_i^+(v)$ and $B_i^0(\mathbf{0}, v)$, etc. \square

Lemma 7. *In an execution α , consider $\mathcal{V}_f = \{v : \exists A_i(v) \in \alpha\}$, then for any $A_j(v)$ step with $j \geq i+3$ in α , it is the case that $v > \min(\mathcal{V}_f)$ or the algorithm decides.*

Proof. The proof is by contradiction and the idea is to apply Lemma 6 on three consecutive adopt-commit-max objects ($C[i]$, $C[i+1]$, and $C[i+2]$) and show that either the algorithm decides or that $v_m (= \min(\mathcal{V}_f))$ does not propagate beyond these three adopt-commit-max objects. Due to Lemma 6 we know that all processes, except p_a , p_b execute continuously for at least four rounds. We also know that operating on an adopt-commit-max object in Archipelago has only three round-steps (R , A , and B). Because of this, after three adopt-commit-max objects, we can show that for adopt-commit-max-object $C[i+2]$, there are r'' and x'' such that a process takes a step $R\langle i+3, \cdot \rangle$ before some $r'' + x'' + 2$, which contradicts Lemma 6.

To prove this lemma, assume by way of contradiction that there is an execution α such that (1) the algorithm does not decide in α , (2) α contains an $A_i(v_m)$ step and (3) α contains an $A_j(v_m)$ step, where $j \geq i+3$.

Due to Lemma 6, we know that if there is a $j \geq i + 3$ with $A_j(v_m)$, then the execution looks like Figure 2.4. Because $x \geq 2$, we have at least 4 continuous suspensions from round $r + 1$ to round $r + x + 2$.

	...	$r-1$	r	$r+1$...	$r+x-1$	$r+x$	$r+x+1$	$r+x+2$	$r+x+3$...
p_a		.	$A_i^0(v_m)$	X	X	X	X	$B_i(1, v_m)$	$R^0(i+1, v_m)$.	
p_b		$A_i^+(v)$	$B_i^0(0, v)$	X	X	.	
p_c		
\vdots											

Figure 2.4 – Lemma 7 (1)

Note, that in any execution, a process takes a sequence of steps: $R\langle i_1, \cdot \rangle, A_{i_1}, B_{i_1}, R\langle i_2, \cdot \rangle, A_{i_2}, B_{i_2}, \dots$ where $i_1 < i_2 < \dots$. We show that all processes must perform certain steps in this sequence prior to certain rounds. One of the three steps that p_c 's takes in rounds $r + 1$, $r + 2$ or $r + 3$ is an R step that returns a value that is at least $\langle i, \cdot \rangle$, since process p_a performed an A_i^0 step in round r . Thus, by round $r + x + 2$, p_c must perform an A_j step with $j \geq i$. Processes p_a and p_b have also performed a step A_i by round $r + x + 2$. So, every process in the system has performed an A_j step with $j \geq i$ by round $r + x + 2$.

By assumption, value v_m does not get eliminated, and hence when the algorithm operates on adopt-commit-max object $C[i + 1]$ we have the exact same execution as in Figure 2.3 but for adopt-commit-max object $C[i + 1]$. See Figure 2.5. Again, let $p_{a'}$ and $p_{b'}$ be the processes described in Lemma 6 with respect to A_{i+1} and let $p_{c'}$ be any other process. Note that in process $p_{a'}$ is not necessarily the same as process p_a , etc., since it could be that a different process is the one that performs the $A_{i+1}^0(v_m)$ now. For example, it could be that $p_{a'} = p_c$ and $p_{b'} = p_a$. Also, note that round numbers are now based upon $r' \neq r$. By Lemma 6, no $R\langle i + 1, \cdot \rangle$ occurs before round $r + x + 2$ and since $p_{a'}$ does an $R\langle i + 1, \cdot \rangle$ step before round r' , we have $r' > r + x + 2$. Thus, $p_{c'}$ must perform a step A_j with $j \geq i$ before round r' . Then, $p_{c'}$ takes at least four more steps by round $r' + x' + 2$. So, $p_{c'}$ must perform a step B_k with $k \geq i + 1$ by round $r' + x' + 2$. Processes $p_{a'}$ and $p_{b'}$ have performed step B_{i+1} by round $r' + x' + 2$. So, every process performs a step B_k with $k \geq i + 1$ by round $r' + x' + 2$.

Again, because of Lemma 6, this pattern of execution should appear for adopt-commit-max object $C[i + 2]$. Consider Figure 2.6. Again, let $p_{a''}$ and $p_{b''}$ be the processes described in Lemma 6 with respect to A_{i+2} and let $p_{c''}$ be any other process. By Lemma 6, no $R\langle i + 2, \cdot \rangle$ step occurs before round $r' + x' + 2$ and since process $p_{a''}$ does such a step before round r'' , we have $r'' > r' + x' + 2$. Thus, $p_{c''}$ must perform a step B_k with $k \geq i + 1$ before round r'' . Then, $p_{c''}$ takes at least four more steps by round $r'' + x'' + 2$. Hence, by round $r'' + x'' + 2$, $p_{c''}$ must perform a step $R\langle \ell, \cdot \rangle$ with $\ell \geq i + 3$. This contradicts the fact that no $R\langle i + 3, \cdot \rangle$ step occurs

	...	$r'-1$	r'	$r'+1$...	$r'+x'-1$	$r'+x'$	$r'+x'+1$	$r'+x'+2$	$r'+x'+3$...
$p_{a'}$.	$A_{i+1}^0(v_m)$	X	X	X	X	$B_{i+1}(\mathbf{1}, v_m)$	$R^0(i+2, v_m)$.	
$p_{b'}$		$A_{i+1}(v)$	$B_{i+1}^0(\mathbf{0}, v)$	X	X	.	
$p_{c'}$		
\vdots											

Figure 2.5 – Lemma 7 (2)

	...	$r''-1$	r''	$r''+1$...	$r''+x''-1$	$r''+x''$	$r''+x''+1$	$r''+x''+2$	$r''+x''+3$...
$p_{a''}$.	$A_{i+2}^0(v_m)$	X	X	X	X	$B_{i+2}(\mathbf{1}, v_m)$	$R^0(i+3, v_m)$.	
$p_{b''}$		$A_{i+2}^*(v)$	$B_{i+2}^0(\mathbf{0}, v)$	X	X	.	
$p_{c''}$		
\vdots											

Figure 2.6 – Lemma 7 (3)

before step $r'' + x'' + 2$ dictated by Lemma 6.

□

Lemma 7 implies Theorem 1, because either the algorithm decides or the minimum value proposed to an adopt-commit-max object $C[i]$ does not propagate in any later adopt-commit-max object $C[j]$ with $j \geq i + 3$. Hence, due to the continual elimination of the current minimal value, eventually only one value gets proposed to an adopt-commit-max object and hence the algorithm decides. Finally, note that if we had devised Archipelago for binary consensus, this would not substantially simplify the proof. We would still need to prove that the minimum value, in this case 0, does not propagate in later adopt-commit objects.

Termination. Archipelago satisfies *termination* for $n \geq 3$, meaning that in an \diamond synchronous execution, every correct process eventually decides. In such an execution, Archipelago needs at most 5 rounds, after processes have been aligned in rounds. This alignment can take place after the global stabilization time (GST) round [48]. Note that after the GST round, it is not guaranteed that the rounds are aligned (i.e., all processes start and end a round at the same time). Therefore, after the GST round, the rounds need to be aligned first. When the alignment occurs, Archipelago needs at most 5 rounds to decide.

Specifically, we show that Archipelago terminates in any \diamond synchronous execution with up to $f = n - 1$ crashes. Consider such an execution and let r be a round such that (1) the system

has reached synchrony by round r and (2) each process p is either correct or p has crashed by round r . In such an \diamond synchronous execution, Archipelago needs at most 5 rounds starting from round r in order to decide.

As in the proof of leaderless termination for Archipelago, we assume a model with $n \geq 3$ processes. In this scenario, since processes take steps without crashes starting from round r , every correct process p takes steps R , A , and B without suspensions somewhere between round r and $r + 5$. Each process p performs an R step at least by round $r + 2$, because p can perform step A in round r and then B in round $r + 1$. Consider a process p that performs an $R^0\langle i, v \rangle$ step with the greatest $\langle i, v \rangle$ value. This means, that p immediately afterwards performs $A_i^0(v)$ and then $B_i^0(1, v)$ and due to Lemma 3 Archipelago decides. If multiple such processes perform $R^0\langle i, v \rangle$, then all the processes retrieve the same maximum value $\langle i, v \rangle$ from m (Line 24) and hence propose the same value to adopt-commit-max object $C[i]$ and perform steps $A_i^0(v)$ and $B_i^0(1, v)$ and hence the algorithm decides (see Lemma 3).

The above discussion implies that Archipelago satisfies termination, thus meaning that in an \diamond synchronous execution, Archipelago decides. Furthermore, note that the Archipelago can withstand up to $f = n - 1$ crashes and decides in an \diamond synchronous execution. Naturally, the message passing variant of Archipelago (Section 2.6) can only withstand up to $f = (n - 1)/2$ crashes.

2.6 Leaderless Consensus in Message Passing

In this section we show how our shared memory results translate to the message passing model. The definition of \diamond synchronous-1 in message passing is the same as in shared memory: an execution is \diamond synchronous-1 if it is equal to a sequence of *rounds*, such that at most 1 process can be suspended in each round. We say that a process p is suspended [10] in a round r , if p does not send any messages in r and does not receive any messages sent by other processes in round r .

Additionally, we assume that up to $f - 1$ (where, for simplicity, $n = 2f + 1$) processes may fail by crashing forever.²

We use the following notion of round: in each round r , every (correct, non-suspended) process p_i (i) broadcasts a message (we call this first message a *request*), (ii) delivers all requests that were sent to p_i in r , (iii) sends a message (we call this second message a *response*) for every request it has delivered in (ii), and (iv) delivers all replies sent to it in r . Note that this notion

²We allow up to $f - 1$ crashes (instead of the more typical f) so that, taking into account at most one additional suspended process per round, we can use the familiar $f + 1$ quorum size.

of round involves 2 message delays. With this definition of $\diamond\text{synchronous-}k$, the leaderless termination property remains the same as in Definition 1.

It is easy to see that Paxos (in message-passing) is not leaderless. With our notion of round, the prepare and propose phases correspond to a round each. At each round, the adversary selects the proposer p that has completed the prepare phase with the highest ballot number so far and suspends p in its propose phase.

The Crash Case. In order to obtain a leaderless consensus algorithm in a message-passing system with crash failures, one might be tempted to apply the ABD emulation [13] to Algorithm 4. However, this ABD-emulated version of Archipelago would require at least two message-passing rounds per step (R , A and B): one round for the write and one round for the collect (n reads in parallel). It is unclear whether this emulated algorithm would be leaderless, since our Archipelago's proof hinges on each step (R , A and B) taking exactly one round.

Instead, we obtain a message-passing version of Archipelago (shown in Algorithm 5) by combining the write and the collect in a single round. The broadcast in lines 16, 24, and 31 acts as both the write invocation and the read invocation. Processes' responses in lines 42, 46, and 50 serve to confirm the write, and return all values written so far.

Algorithm 5 Archipelago in message passing with $n = 2f + 1$

```

1:  $\triangleright$  Local State
2:  $i$ , the current adopt-commit-max object, initially 0
3:  $R$ , a set of tuples, initially empty
4:  $A[0, 1, \dots]$ , a sequence of sets, all initially empty
5:  $B[0, 1, \dots]$ , a sequence of sets, all initially empty
6:
7: procedure propose( $v$ )
8:   while true do
9:      $\langle i, v' \rangle \leftarrow \text{R-Step}(v)$ 
10:     $\langle \text{flag}, v'' \rangle \leftarrow \text{A-Step}(v')$ 
11:     $\langle \text{control}, \text{val} \rangle \leftarrow \text{B-Step}(\text{flag}, v'')$ 
12:    if  $\text{control} = \text{commit}$  then return  $\text{val}$ 
13:    else  $i \leftarrow i + 1$ 
14:
15: procedure R-Step( $v$ )
16:   broadcast( $R, i, v$ )
17:   wait until receive (R-response,  $i, R$ ) from  $f + 1$  processes
18:
19:    $R \leftarrow R \cup \{\text{union of all } R\text{s received in Line 17}\}$ 
20:    $\langle i', v' \rangle \leftarrow \text{max}(R)$   $\triangleright$  note that  $\langle x, y \rangle > \langle x', y' \rangle$  if  $x > x'$  or  $(x = x' \text{ and } y > y')$ 
21:   return  $\langle i', v' \rangle$ 
22:
23: procedure A-Step( $v$ )
24:   broadcast( $A, i, v$ )
25:   wait until receive (A-response,  $i, A[i]$ ) from  $f + 1$  processes
26:
27:    $\mathcal{S} \leftarrow \text{union of all } A[i]\text{s received}$ 
28:   if  $\mathcal{S}$  contains only one value  $\text{val}$  then return  $\langle \text{true}, \text{val} \rangle$ 
29:   else return  $\langle \text{false}, \text{max}(\mathcal{S}) \rangle$ 

```

```
30: procedure B-Step( $flag, v$ )
31:   broadcast( $B, i, flag, v$ )
32:   wait until receive (B-response,  $i, B[i]$ ) from  $f + 1$  processes
33:
34:    $\mathcal{S} \leftarrow$  union of all  $B[i]$ s received
35:   if  $\mathcal{S}$  contains only  $\langle \text{true}, val \rangle$  for some  $val$  then
36:     return  $\langle \text{commit}, val \rangle$ 
37:   else if  $\mathcal{S}$  contains some entry  $\langle \text{true}, val \rangle$  then return  $\langle \text{adopt}, val \rangle$ 
38:   else return  $\langle \text{adopt}, v \rangle$ 
39:
40: upon event reception of  $(R, j, v)$  from  $p$  do
41:   Add  $\langle j, v \rangle$  to  $R$ 
42:   send(R-response,  $j, R$ ) to  $p$ 
43:
44: upon event reception of  $(A, j, v)$  from  $p$  do
45:   add  $v$  to  $A[j]$ 
46:   send(A-response,  $j, A[j]$ ) to  $p$ 
47:
48: upon event reception of  $(B, j, flag, v)$  from  $p$  do
49:   add  $\langle flag, v \rangle$  to  $B[j]$ 
50:   send(B-response,  $j, B[j]$ ) to  $p$ 
```

This way of combining writes and reads can break atomicity, but is sufficient to guarantee safety (of consensus) during asynchronous periods. More precisely, the R-Step behaves like a “regular” max-register, one that returns valid, non-decreasing values to each invoker (see Lemma 8), and the A- and B-Steps together behave like an adopt-commit object (see Lemma 9). As such, our proof of safety in Section 2.5 applies to Algorithm 5 as well.

The non-atomic behavior exhibited during asynchronous periods is due to the overlap in time of the request and response parts of each round. However, during synchronous-1 periods, we can assume that requests are delivered by all processes before any response is sent out. Thus, once the system becomes permanently synchronous-1, the R-Step satisfies the (atomic) max-register properties and the A- and B-Steps together behave like an adopt-commit-max object. Therefore, our proof of leaderless termination in Section 2.5 remains valid for Algorithm 5 as well.

In what follows, we provide some lemmas that show the safety of Algorithm 5. Note that all results and line numbers refer to Algorithm 5.

Lemma 8. *The R-Step satisfies the following properties:*

Validity For a fixed i , if some process returns v , then v was the input of some process.

Monotonicity If process p returns (i, v_i) in an R-Step and p returns (j, v_j) in a later R-Step, then $j \geq i$ and $v_j \geq v_i$.

Proof.

Validity At Line 20 (i, v') (the value returned by the R-Step) is computed as the maximum of all tuples ever received, which must in turn have been broadcast at Line 16 by some process.

Monotonicity Assume by contradiction that some process p returns (i, v_i) in R-Step r_1 and later returns (j, v_j) in R-Step r_2 such that $(j, v_j) < (i, v_i)$. During r_1 , p selected and returned (i, v_i) as the maximum element of its local R set. Since elements can only be appended to a process's R set, (i, v_i) will still be in R during r_2 . Thus, p cannot select and return a tuple smaller than (i, v_i) during r_2 . We have reached a contradiction.

□

Lemma 9. *For a fixed i , an A-Step followed by a B-Step corresponds to an adopt-commit object.*

Proof. Validity holds because at lines 28, 29, 36, 37, and 38, processes only return values that were sent at lines 46 or 50. In turn, these values must be input values of some process who broadcast them at lines 24 or 31.

Termination holds because the only waiting is done at lines 25 and 32; processes always wait for $f + 1$ responses; since $f + 1 = n - f$, processes eventually receive these responses.

Commitment holds because if all processes enter A-Step with the same value v , then the check at Line 28 will succeed and all processes will enter B-Step with (true, v) ; thus the check at Line 35 will succeed and all processes will return (commit, v) in the B-Step.

Agreement. Assume by contradiction that process p outputs (commit, v) and process p' outputs (\cdot, v') with $v \neq v'$. Then p must have received B-responses containing only (true, v) from a set R_p of $f + 1$ distinct processes; p' must have also received B-responses from a set $R_{p'}$ of $f + 1$ distinct processes. Since $f + 1 > n/2$, R_p and $R_{p'}$ must intersect in at least one process q .

Let \mathcal{S} be the union of all $B[i]$ s received by p' in B-responses. We distinguish three cases, based on the number of distinct values val for which the \mathcal{S} contains (true, val) .

- \mathcal{S} does not contain any $(\text{true}, \text{val})$ tuples. In this case, q 's B-response to p' must contain a $(\text{false}, \text{val})$ tuple. If q responded to p before p' , then by Lemma 10 q 's B-response to p' must include a (true, v) tuple — a contradiction. If q responded to p' before p , then by Lemma 10 q 's B-response to p must include $(\text{false}, \text{val})$ — a contradiction.
- \mathcal{S} contains $(\text{true}, \text{val})$ tuples for a single value val . Then $\text{val} \neq v$, otherwise p' would either commit v or adopt v . Assume without loss of generality that q responds to p before it responds to p' . Then q 's response to p' must contain both (true, v) and $(\text{true}, \text{val})$, contradicting Lemma 11.
- \mathcal{S} contains more than one value v . This is impossible by Lemma 11.

□

Lemma 10. *For a fixed i , if a process p sends a B-response (B-response, $i, B[i]$) to some process q at time t and p sends a B-response (B-response, $i, B[i]'$) to some process q' at time $t' > t$, then $B[i] \subseteq B[i]'$.*

Proof. This is because items can only be added to $B[i]$ (Line 49). □

Lemma 11. *For a fixed i , if two processes p and q broadcast (true, v) and (true, v') at Line 31, then $v = v'$.*

Proof. Assume not, then p must have received A-responses containing only v from a set R_p of $f + 1$ processes and q must have received A-responses containing only v' from a set $R_{p'}$ of $f + 1$ processes. Since $f + 1 > n/2$, R_p and $R_{p'}$ must intersect in at least one process r . Assume without loss of generality that r responded to p first and then to q : then the response to q must also include v by Lemma 10. We have reached a contradiction. □

The Byzantine Case. We conjecture that modifying Algorithm 5 in the following way yields a Byzantine-fault tolerant³ leaderless consensus algorithm:

1. The number of processes becomes $n = 3f + 1$.
2. Processes wait for $2f + 1$ (instead of $f + 1$) responses at lines 17, 25 and 32.
3. Replace the max function in lines 20 and 29 with the max- f function; this function takes $2f + 1$ or more arguments, orders them in decreasing order, discards the first f values in this order and returns the first remaining value.

³We refer here to the weak Byzantine agreement version of the problem [76], whose validity property is: *With no faulty processes, if some process decides v , then v is the input of some process.*

4. Replace the B-Step as shown in Algorithm 6.
5. All messages are signed. At each step (R , A and B) except the very first R step, when broadcasting, processes include a *certificate* consisting of all $2f + 1$ responses received in the previous step. Processes only respond to requests with valid certificates, i.e., requests for which v , $flag$ and/or i (depending on the step) were chosen correctly according to the responses in the certificate.
6. Replace broadcast with reliable broadcast.

Algorithm 6 Fragment: B-Step in Byzantine version of Archipelago in message passing where $n = 3f + 1$

```

1: procedure B-Step( $flag, v$ )
2:   broadcast( $B, i, flag, v$ )
3:   wait until receive (B-response,  $i, B[i]$ ) from  $2f + 1$  processes
4:
5:    $\mathcal{S} \leftarrow \{ \text{all } B[i] \text{ s received} \}$ 
6:   if  $f + 1$  sets in  $\mathcal{S}$  contain only  $\langle \text{true}, val \rangle$  for some  $val$  then
7:     return  $\langle \text{commit}, val \rangle$ 
8:   else if  $f + 1$  sets in  $\mathcal{S}$  contain some entry  $\langle \text{true}, val \rangle$  then
9:     return  $\langle \text{adopt}, val \rangle$ 
10:  else return  $\langle \text{adopt}, v \rangle$ 

```

Our changes have the following rationales: (1) and (2) increase the quorum size such that any two quorums intersect in (at least) one correct process. Change (3) serves to eliminate any maliciously-chosen high values in the R- and A-Steps. Note that our proofs in Section 2.5 remain valid if we replace \max by $\max - f$ since both functions discard the minimum value as long as one correct process does not propose the minimum value. Change (4) aims to prevent Byzantine processes from getting an invalid value adopted or committed by sending B-responses fake $\langle \text{false}, \cdot \rangle$ or $\langle \text{true}, \cdot \rangle$ tuples. Change (5) aims to allow honest processes to validate that other processes have correctly processed the responses received at the previous step.

2.7 ArchSMR: Archipelago in Practice

To demonstrate the different impact of failures on leaderless and leader-based consensus algorithms, we implemented *ArchSMR*, a state machine replication (SMR) algorithm based on Archipelago (Algorithm 5), deployed it in a distributed setting and compared its behavior to Apache ZooKeeper [64], a production-level leader-based SMR.

As ZooKeeper is written in Java, we implemented Archipelago in Java and built ArchSMR with an unbounded sequence of separate Archipelago instances just as Multi-Paxos uses a sequence of separate Paxos instances [78]. We deployed ArchSMR along with ZooKeeper on 3 Amazon EC2 *t2.large* instances with 2 vCPUs, 8 GB of memory running Ubuntu Server v18.04, all within the same availability zone.

Specifically, in the following experiments, we have $n = 3$ servers on distinct VMs and 3 clients, each sending requests to and receiving responses from one server. Note that the goal of our experiments is to show that the leaderless algorithm Definition 2, as well as Archipelago can be of practical relevance. We are not interested in the maximum throughput without failures but by the average throughput during failures, so we fixed the sending rate to 500 requests by second (~ 166 requests per second per client) for both SMRs.

In order to experiment the $\diamond\text{synchronous}-1$ model described in Section 2.2, we stop a server by sending a POSIX.1-1990 standard SIGSTOP signal to one server process and later resuming it by sending a POSIX.1-1990 standard SIGCONT signal. In practice, the effects of such a suspension can occur in case of a software bug, CPU overload, overheating, etc. Naturally, our results also hold if we crash and restart a server instead of suspending it.

In what follows, we first consider the case where we suspend the leader server in ZooKeeper and some arbitrary server in ArchSMR. Afterwards, we show how ArchSMR performs when there are recurring suspensions of different servers.

Single Suspension. The main characteristic of a leaderless algorithm is that suspending or crashing one server does not decimate throughput (i.e., drop throughput to 0). Definition 2 precisely captures this characteristic by arguing that irrespectively of which server we suspend, the algorithm is still able to complete operations and hence throughput is not decimated.

To confirm this experimentally, we suspended one server while running ArchSMR and ZooKeeper and sampling their throughput every 100 ms. Figure 2.7 depicts the throughput evolution of ArchSMR and ZooKeeper over time without depicting the warm-up and cool-down phases of the experiments. The suspension, depicted with a vertical solid line, suspends an arbitrary server in ArchSMR, since all servers have the same role, and suspends the leader server in ZooKeeper.

As expected, when we suspend the leader server in ZooKeeper, throughput drops to 0. Note that the throughput remains 0 for more than 3 seconds. This is due to the default configuration (`tickTime` and `initLimit` [66]) of ZooKeeper. When the leader server is suspended, the ZooKeeper cluster needs more than 10 s to initiate a new leader election. In contrast, if we crash the leader server in ZooKeeper, then the socket connection closes and the rest of the servers immediately recognize the leader crash and initiate a leader election. By default,

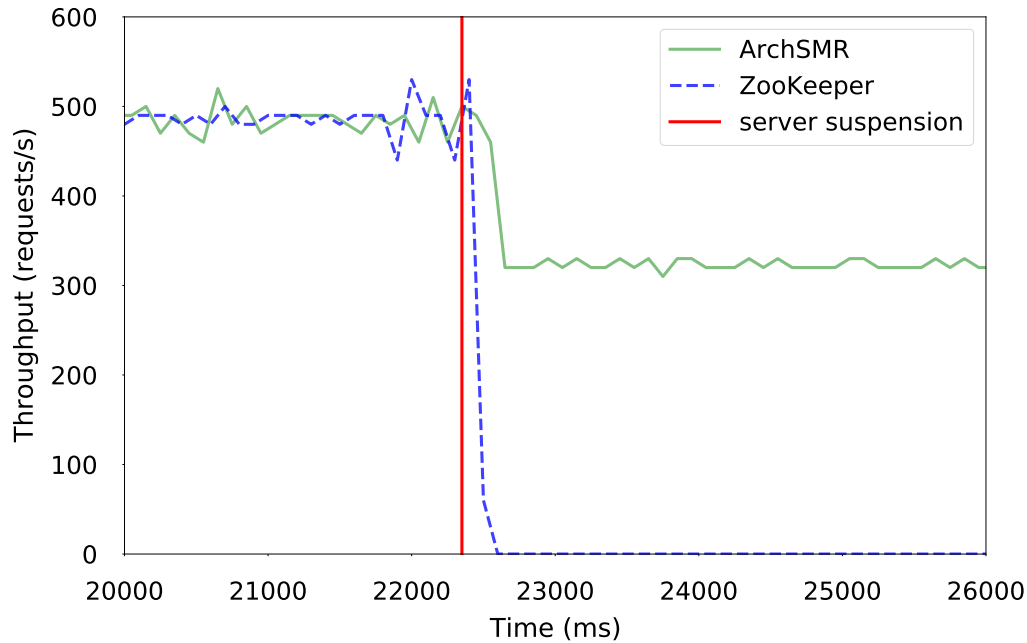


Figure 2.7 – Performance of ZooKeeper and ArchSMR upon server suspension.

ZooKeeper needs at least 200 ms to initiate a new leader election [67] after a leader crash. One can anyway expand ZooKeeper downtime, by suspending a server from the moment it becomes leader until it gets deposed.

In contrast to ZooKeeper, the throughput of ArchSMR simply drops to about 330 requests per second, due to the fact that one of the 3 clients keeps sending around 166 requests per second to the suspended process. This shows that the leaderless algorithm Definition 2 is of practical interest for robustness.

Recurring Suspensions. Definition 1 suggests that a leaderless consensus algorithm decides even if we suspend at most one server at a time. Naturally, we would expect the same to be the case for ArchSMR. Here, we investigate how ArchSMR performs when we have a recurring number of suspensions.

Figure 2.8 depicts the throughput of ArchSMR during recurring suspensions. Specifically, in a round-robin fashion, we suspend each server for two seconds and then wait for one second before suspending the next server, etc. We wait for one second to guarantee that we have all servers up and running. As can be seen in Figure 2.8, throughput drops by one third during the suspension of a server and when the server gets unsuspended we have a jump in throughput

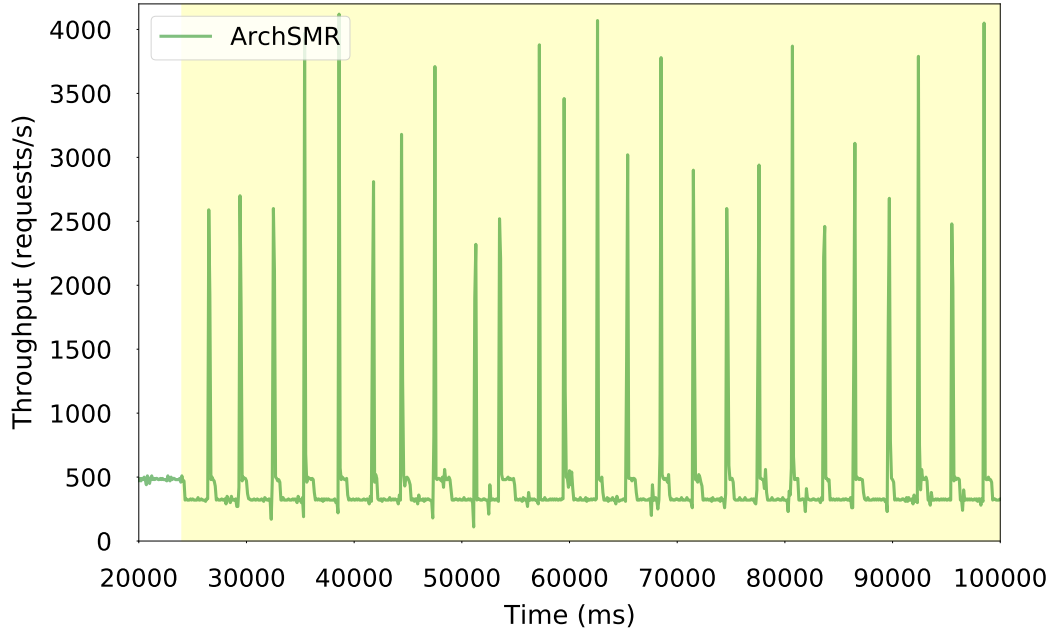


Figure 2.8 – Performance of ArchSMR upon rotating suspensions of all servers.

since the previously issued commands of the client get decided. Notice that even though throughput drops when a server is suspended, the average throughput (e.g., from 20 s to 30 s) in the shaded area of the plot remains at around 480 requests per second. So, even though, throughput drops, the average throughput remains almost the same when considering no suspensions.

2.8 Related Work

Given that the limitation of a leader in consensus or SMR is well-known [6, 15, 30, 41, 57, 60, 64, 91, 96, 121, 125], it is surprising to see that the notion of leaderless has never been defined. But with the recent need to scale SMRs to blockchain networks [31, 35, 41, 121, 125], this limitation is exacerbated. Crain et al. [41] proposed DBFT for blockchains. It relies on n binary consensus instances, each using a weak coordinator to converge even if sufficiently many correct processes propose distinct values. DBFT is not leaderless according to our definition as the same weak coordinator can be used by all instances. Maofan et al. [125] replaced the leader's large proposals of PBFT [35] by smaller message digests to obtain HotStuff. HotStuff throughput still drops to zero when the leader fails and until some view-change completes [121]. These observations serve as motivations for our work.

In a brief announcement [81], Lamport proposed a high level transformation of a class of leader-based consensus algorithms into a class of leaderless algorithms using repeatedly a synchronous virtual leader election algorithm where all processes try to agree on a set of proposals. In a corresponding patent document [82], Lamport explains that during a period of asynchrony, if the virtual leader election fails, then the consensus algorithm may not progress but should not violate safety as long as it tolerates malicious leaders [81]. The adopt-commit-max object of Archipelago allows processes to converge towards a unique value, hence sharing similarities with the proposal of some virtual leader. Yet, neither a leaderless definition nor a virtual leader specification were given.

Borran and Schiper proposed a so-called “leader-free” consensus algorithm [38] without offering a leader-freedom definition. The algorithm has an exponential complexity, which limits its applicability.

In randomized consensus algorithms (e.g., [20, 98, 104]), correct processes typically execute the same series of steps, but they do not converge to the same value deterministically.

Interestingly, SMR algorithms that rely on multiple leaders (e.g., Mencius [91]) do not necessarily rely on a leaderless consensus algorithm. Moraru et al. [96] used multiple “command leaders” in EPaxos. Each command leader commits one command as long as commands are compatible. However, in the general case, where commands have dependencies, then only one of the command leaders can get its command committed at a time, as if there were successive leader-based consensus instances. If a leader fails after receiving a positive acknowledgement from a fast quorum of $n - 1$ processes, it rejoins with a new identifier and a greater ballot without being able to acknowledge the previous commit message. Despite being specified in TLA+, EPaxos specification was recently shown incorrect [115], indicating that designing a multi-leader algorithm is error prone.

2.9 Conclusion

In this chapter, we defined what it means for a consensus algorithm to be leaderless. To the best of our knowledge, this work is the first to formally introduce the notion of a leaderless consensus algorithm. Then, we devised Archipelago, a leaderless consensus algorithm. We proved that Archipelago is correct, a daunting task since we had to prove that in any synchronous-1 execution that starts from an arbitrary initial state, Archipelago decides. We then translated our result to the message-passing model. Afterwards, we conjectured that Archipelago can be modified to tolerate Byzantine failures. Finally, we used Archipelago to build a state machine replication algorithm, and illustrated its robustness in a failure scenario against ZooKeeper.

3 State Machine Replication is More Expensive Than Consensus

Consensus and State Machine Replication (SMR) are generally considered to be equivalent problems. In certain system models, indeed, the two problems are computationally equivalent: any solution to the former problem leads to a solution to the latter, and vice versa.

In this chapter, we study the relation between consensus and SMR from a *complexity* perspective. We find that, surprisingly, completing an SMR command can be more expensive than solving a consensus instance. Specifically, given a synchronous system model where every instance of consensus always terminates in constant time, completing an SMR command does *not* necessarily terminate in constant time. Besides theoretical interest, our result also corresponds to practical phenomena we identify empirically. We experiment with two well-known SMR implementations (Multi-Paxos and Raft) and show that, indeed, SMR is more expensive than consensus in practice. One important implication of our result is that—even under synchrony conditions—no SMR algorithm can ensure bounded response times.

3.1 Introduction

Consensus is a fundamental problem in distributed computing. In this problem, a set of distributed processes need to reach agreement on a single value [77]. Solving consensus is one step away from implementing State Machine Replication (SMR) [75, 110]. Essentially, SMR consists of replicating a sequence of commands—often known as a log—on a set of processes which replicate the same state machine. These commands represent the ordered input to the state machine. SMR has been successfully deployed in applications ranging from storage systems, e.g., LogCabin built on Raft [100], to lock [36] and coordination [64] services. At a high level, SMR can be viewed as a sequence of consensus instances, so that each value output from an instance corresponds to a command in the SMR log.

From a *solvability* standpoint and assuming no malicious behavior, SMR can use consensus

as a building block. When the latter is solvable, the former is solvable as well (the reverse direction is straightforward). Most previous work in this area, indeed, explain how to build SMR assuming a consensus foundation [53, 78, 83], or prove that consensus is equivalent from a solvability perspective with other SMR abstractions, such as atomic broadcast [37, 99]. An important body of work also studies the complexity of individual consensus instances [54, 68, 80, 108]. SMR is typically assumed to be a repetition of consensus instances [69, 79], so at first glance it seems that the complexity of an SMR command can be derived from the complexity of the underlying consensus. We show that this is not the case.

In practice, SMR algorithms can exhibit irregular behavior, where some commands complete faster than others [34, 96, 123]. This suggests that the complexity of an SMR command can vary and may not necessarily coincide with the complexity of consensus. Motivated by this observation, we study the relation between consensus and SMR in terms of their *complexity*. To the best of our knowledge, we are the first to investigate this relation. In doing so, we take a formal, as well as a practical (i.e., experimental) approach. Counter-intuitively, we find that SMR is not necessarily a repetition of consensus instances.

We show that completing an SMR command can be more expensive than solving a consensus instance. Constructing a formalism to capture this result is not obvious. We prove our result by considering a fully synchronous system, where every consensus instance always completes in a *constant number of rounds*, and where at most one process in a round can be suspended (e.g., due to a crash or because of a network partition). A suspended process in a round is unable to send or deliver any messages in that round. Surprisingly, in this system model, we show that it is impossible to devise an SMR algorithm that can complete a command in constant time, i.e., completing a command can potentially require a *non-constant number of rounds*. We also discuss how this result applies in weaker models, e.g., partially synchronous, or if more than one process is suspended per round (see Section 3.3.2).

At a high level, the intuition behind our result is that a consensus instance “leaks,” so that some processing for that instance is deferred for later. Simply put, even if a consensus instance terminates, some protocol messages belonging to that instance can remain undelivered. Indeed, consensus usually builds on majority quorum systems [117], where a majority of processes is sufficient and necessary to reach agreement; any process which is not in this majority may be left out. Typically, undelivered messages are destined to processes which are not in the active majority—e.g., because they are slower, or they are partitioned from the other processes. Such a leak is inherent to consensus: the instance must complete after gathering a majority, and should not wait for additional processes. If a process is not in the active majority, that process might as well be faulty, e.g., permanently crashed.

In the context of an SMR algorithm, when successive consensus instances leak, the same process can be left behind across multiple SMR commands; we call this process a *straggler*.

Consequently, the deferred processing accumulates. It is possible, however, that this straggler is in fact correct. This means that eventually the straggler can become part of the active quorum for a command. This can happen when another process fails and the quorum must switch to include the straggler. When such a switch occurs, the SMR algorithm might not be able to proceed before the straggler recovers the whole chain of commands that it misses. Only after this recovery completes can the next consensus instance (and SMR command) start. Another way of looking at our result is that a consensus instance can neglect stragglers, whereas SMR must deal with the potential burden of helping stragglers catch-up.¹

We experimentally validate our result in two well-known SMR systems: a Multi-Paxos implementation (LibPaxos [3]) and a Raft implementation (etcd [2]). Our experiments include the wide-area and clearly demonstrate the difference in complexity, both in terms of latency and number of messages, between a single consensus instance and an SMR command. Specifically, we show that even if a single straggler needs to be included in an active quorum, SMR performance noticeably degrades. It is not unlikely for processes to become stragglers in practical SMR deployments, since these algorithms typically run on commodity networks [19]. These systems are subject to network partitions, processes can be slow or crashed, and consensus-based implementations can often be plagued with corner-cases or implementation issues [22, 36, 62, 73], all of which can lead to stragglers.

Our result—that an SMR algorithm cannot guarantee a constant response time, even if otherwise the system behaves synchronously—brings into focus a trade-off in SMR. In a nutshell, this is the trade-off between the best-case performance and the worst-case performance of an SMR algorithm. On the one hand, such an algorithm can optimize for the worst-case performance. In this case, the algorithm can dedicate resources (e.g., by provisioning additional processes or assisting stragglers) to preserve its performance even when faults manifest, translating into lower tail latencies; there are certain classes of SMR-based applications where latencies and their variability are very important [11, 40, 42]. On the other hand, an SMR algorithm can optimize for best-case performance, i.e., during fault-free periods, so that the algorithm advances despite stragglers being left arbitrarily behind [63, 96]. This strategy means that the algorithm can achieve superior throughput, but its performance will be more sensible to faults.

The contribution we present in this chapter is twofold. First, we initiate the study of the relation, in terms of complexity, between consensus and SMR. We devise a formalism to capture the difference in complexity between these two problems, and use this formalism to prove that completing a single consensus instance is not equivalent to completing an SMR command in terms of their complexity (i.e., number of rounds). More precisely, we prove that

¹We note that this leaking property seems not only inherent in consensus, but in any equivalent replication primitive, such as atomic broadcast.

it is impossible to design an SMR algorithm that can complete a command in constant time, even if consensus always completes in constant time. Second, we experimentally validate our theoretical result using two SMR systems in both a single-machine and a wide-area network.

Roadmap. The rest of this chapter is organized as follows. We describe our system model in Section 3.2. In Section 3.3 we present our main result, namely that no SMR algorithm can complete every command in a constant number of rounds. Section 3.4 presents experiments to support our result. We describe the implications of our result in Section 3.5, including ways to circumvent it and a trade-off in SMR. Finally, Section 3.6 concludes this chapter.

3.2 Model

This chapter studies the relation in terms of complexity between consensus and State Machine Replication (SMR). In this section we formulate a system model that enables us to capture this relation, and also provide background notions on consensus and SMR.

We consider a synchronous model and assume a finite and fixed set of processes $\Pi = \{p_1, p_2, \dots, p_n\}$, where $|\Pi| = n \geq 3$. Processes communicate by exchanging messages. Each message is taken from a finite set $M = \{m_1, \dots\}$, where each message has a positive and a bounded size, which means that there exists a $B \in \mathbb{N}^+$ such that $\forall m \in M, 0 < |m| \leq B$.

A process is a state machine that can change its state as a consequence of delivering a message or performing some local computation. Each process has access to a read-only global clock, called *round number*, whose value increases by one on every round. In each round, every process p_i : (1) sends one message to every other process $p_j \neq p_i$ (in total p_i sends $n - 1$ messages in each round);² (2) delivers any messages sent to p_i in that round; and (3) performs some local computation.

An *algorithm* in such a model is the state machine for each process and its initial state. A *configuration* corresponds to the internal state of all processes, as well as the current round number. An *initial configuration* is a configuration where all processes are in their initial state and the round number is one. In each round, up to $n(n - 1)$ messages are transmitted. More specifically, we denote a transmission as a triplet (p, q, m) where $p, q \in \Pi (p \neq q)$ and $m \in M$. For instance, transmission $(p_i, p_j, m_{i,j})$ captures the sending of message $m_{i,j}$ from process p_i to process p_j . We associate with each round an *event*, corresponding to the set of transmissions which take place in that round; we denote this event by $\tau \subseteq \{(p_i, p_j, m_{i,j}) : i, j \in \{1, \dots, n\} \wedge i \neq j\}$. An *execution* corresponds to an alternating sequence of configurations and events, starting with an initial configuration. An execution e^+ is called an *extension* of a finite execution e if e

²As a side note, if a process p_i does not have something to send to process p_j in a given round, we simply assume that p_i sends an empty message.

is a prefix of e^+ . Given a finite execution e , we denote with $E(e)$ the set of all extensions of e . We assume deterministic algorithms: the sequence of events uniquely defines an execution.

Failures. Our goal is to capture the *complexity*—i.e., cost in terms of number of synchronous rounds—of a consensus instance and of an SMR command, and expose any differences in terms of this complexity. Towards this goal, we introduce a failure mode which omits *all* transmissions to and from at most one process per round.

We say that a process p_i is *suspended in round r* associated with the event τ , if $\forall m \in M$ and $\forall j \in \{1, \dots, n\}$ with $j \neq i$, $(p_i, p_j, m) \notin \tau$ and $(p_j, p_i, m) \notin \tau$, hence $|\tau| = n(n-1) - 2(n-1) = (n-1)(n-2)$. If a process p_i is not suspended in a round r , we say that p_i is *correct in round r* . In a round associated with an event τ where all processes are correct there are no omissions, hence $|\tau| = n(n-1)$. A process p_i is *correct* in a finite execution e if there is a round in e where p_i is correct. Process p_i is *correct* in an infinite execution e if there are infinite many rounds in e where p_i is correct. For our result, it suffices that in each round a single process is suspended. Note that each round in our model is a communication-closed layer [49], so messages omitted in a round are not delivered in any later round. This form of suspension is similar to the one presented in the model (Section 2.2) of Chapter 2.

A suspended process represents a scenario where a process is slowed down. This may be caused by various real-world conditions, e.g., a transient network disconnect, a load imbalance, or temporary slowdown due to garbage collection. In all of these, after a short period, connections are dropped and message buffers are reclaimed; such conditions can manifest as message omissions. The notion of being suspended also represents a model where processes may crash and recover, where any in-transit messages are typically lost.

There is a multitude of work [23, 105, 106, 108, 109] on message omissions (e.g., due to link failures) in synchronous models. Our system model is based on the mobile faults model [105]. Note however that our model is stronger than the mobile faults model, since we consider that either exactly zero or exactly $2(n-1)$ message omissions occur in a given round.³ Other powerful frameworks, such as layered analysis [97], the heard-of model [38], or RRFD [52] can be used to capture omission failures, but we opted for a simpler approach that can specifically express the model which we consider.

3.2.1 Consensus

In the consensus problem, processes have initial values which they propose, and have to decide on a single value. Consensus [33] is defined by three properties: validity, agreement, and termination. Validity requires that a decided value was proposed by one of the processes,

³If a process p is suspended, then $n-1$ messages sent by p and $n-1$ messages delivered to p are omitted.

whilst agreement asks that no two processes decide differently. Finally, termination states that every correct process eventually decides. In the interest of having an “apples to apples” comparison with SMR commands (defined below, Section 3.2.2), we introduce a client (e.g., learner in Paxos terminology [78]), and say that a consensus instance completes as soon as the client learns about the decided value. This client is not subject to being suspended, and after receiving the decided value, the client broadcasts this value to the other processes. Algorithm 7 is a consensus algorithm based on this idea.

It is easy to see that in such a model consensus completes in two rounds: processes broadcast their input, and every process uses some deterministic function (e.g., maximum) to decide on a specific value among the set of values it delivers. Since all processes deliver exactly the same set of $n - 1$ (or n) values, they reach agreement. In the second round, all processes which decided send their decided value to all the other processes, including the client. Since $n \geq 3$ and at least $n - 1$ processes are correct in the second round, the client delivers the decided value and thus the consensus instance completes by the end of round two. Afterwards (starting from the third round), the client broadcasts the decided value to all the processes, so eventually every correct process decides, satisfying termination. Note that if a process is suspended in the first round (but correct in the second round), it will decide in the second round, after delivering the decided value from the other processes. Algorithm 7 represents this solution.

We remark that Algorithm 7 does not contradict the lossy link impossibility result of Santoro and Widmayer [105], even though our model permits more than $n - 1$ message omissions in a round, since the model we consider is stronger.

Algorithm 7 Consensus

```

1: procedure propose( $p_i, v_i$ )  $\triangleright p_i$  proposes value  $v_i$ 
2:    $\triangleright$  round 1
3:    $decision \leftarrow \perp$ 
4:    $\forall p \in \Pi \setminus \{p_i\}, \text{send}(p, v_i)$   $\triangleright \Pi$  is the set of processes
5:    $values \leftarrow \{v_i\} \cup \{ \text{each value } v \text{ delivered from process } p \mid \forall p \in \Pi \setminus \{p_i\} \}$ 
6:   if  $|values| \neq 1$  then  $\triangleright p_i$  is correct in round 1
7:      $decision \leftarrow \text{deterministicFunction}(values)$ 
8:   else  $\triangleright p_i$  was suspended
9:      $\triangleright p_i$  cannot decide yet
10:
11:  $\triangleright$  round  $k$  ( $k \geq 2$ ): consensus instance completes in round 2
12: if  $decision \neq \perp$  then  $\triangleright p_i$  knows the decided value
13:    $\forall p \in (\Pi \setminus \{p_i\}) \cup \{client\}, \text{send}(p, decision)$   $\triangleright$  broadcast decided value
14: else
15:    $decision \leftarrow \text{received decision}$   $\triangleright$  message received from other process
  
```

We emphasize that although correct processes can decide in the first round, we consider that

the consensus instance *completes* when the client delivers the decided value. Hence, the consensus instance in Algorithm 7 completes in the second round. In more practical terms, this consensus instance has a constant cost.

3.2.2 State Machine Replication

The SMR approach requires a set of processes (i.e., replicas) to agree on an ordered sequence of commands [75, 110]. We use the terms replica and process interchangeably. Informally, each replica has a log of the commands it has performed, or is about to perform, on its copy of the state machine.

Log. Each replica is associated with a sequence of decided and known commands which we call the *log*. The commands are taken from a finite set $C = \{c_1, \dots, c_k\}$. We denote the log with $\ell(e, p)$ where e is a finite execution, p is a replica, and each element in $\ell(e, p)$ belongs to the set $C \cup \{\epsilon\}$. Specifically, $\ell(e, p)$ corresponds to commands known by replica p after all the events in a finite execution e have taken place (e.g., $\ell(e, p) = c_{i_1}, \epsilon, c_{i_3}$). For $1 \leq i \leq |\ell(e, p)|$, we denote with $\ell(e, p)_i$ the i -th element of sequence $\ell(e, p)$. If there is an execution e and $\exists p \in \Pi$ and $\exists i \in \mathbb{N}^+$ such that $\ell(e, p)_i = \epsilon$, this means that replica p does not have knowledge of the command for the i -th position in its log, while at least one replica does have knowledge of this command. We assume that if a process knows about a command c , then c exists in $\ell(e, p)$. To keep our model at a high-level, we abstract over the details of how each command appears in the log of each replica, since this is typically algorithm-specific. Additionally, state-transfer optimizations or snapshotting [100] are orthogonal to our discussion.

An SMR algorithm is considered *valid* if the following property is satisfied for any finite execution e of that algorithm: $\forall p, p' \in \Pi$ and for every i such that $1 \leq i \leq \min(|\ell(e, p)|, |\ell(e, p')|)$, if $\ell(e, p)_i \neq \ell(e, p')_i$ then either $\ell(e, p)_i = \epsilon$ or $\ell(e, p')_i = \epsilon$. In other words, consider a replica p which knows a command for a specific log position i , i.e., $\ell(e, p)_i = c_k$, where $c_k \in C$. Then for the same log position i , any other process p' can either know command c_k (i.e., $\ell(e, p')_i = c_k$), not know the command (i.e., $\ell(e, p')_i = \epsilon$), or have no information regarding the command (i.e., $|\ell(e, p')| < i$). Note that we only consider valid SMR algorithms.

In what follows, we define what it means for a replica to be a *straggler*, as well as how replicas first learn about commands.

Stragglers. Intuitively, stragglers are replicas that are missing commands from their log. More specifically, let L be $\max_p |\ell(e, p)|$. We say that q is a k -*straggler* if the number of non- ϵ elements in $\ell(e, q)$ is at most $L - k$. A replica p is a *straggler* in an execution e if there exists a $k \geq 1$ such that p is a k -straggler. Otherwise, we say that the replica is a *non-straggler*. A replica that is suspended for a number of rounds could potentially miss commands and hence

become a straggler.

Client. Similar to the consensus client, there is a client process in SMR as well. In SMR, however, the client proposes commands. The client acts like the $(n + 1)$ -th replica in a system with n replicas and its purpose is to supply one command to the SMR algorithm, wait until it receives (i.e., delivers) a response for the command it sent, then send another command, etc. A client, however, is different from the other replicas, since an SMR algorithm has no control over the state machine operating in the client and the client is never suspended. A client operates in lock-step⁴ as follows:

- sends a command $c \in C$ to all the n replicas in some round r ;
- waits until some replica responds to the client's command (i.e., the response of applying the command).⁵

A replica p can respond to a client command c only if it has all commands preceding c in its log. This means that $\exists i : \ell(e, p)_i = c$ and $\forall j < i, \ell(e, p)_j \neq \epsilon$. We say that the client is *suggesting* a command c at a round r if the client sends a message containing command c to all the replicas in round r . Similarly, we say that a client *gets a response* for command c at a round r if some replica sends a message to the client containing the response of the command in round r .

State Machine Replication Algorithm

Algorithm 7 shows that consensus is solvable in our model. It seems intuitive that SMR is solvable in our model as well. To prove that this is the case, we introduce an SMR algorithm for our model (Section 3.2). Roughly speaking, this algorithm operates as follows. Each replica contains an ordered log of decided commands. A command is decided for a specific log position by executing a consensus instance similar to Algorithm 7. The SMR algorithm takes care of stragglers through the use of helping. Specifically, each replica tries to help stragglers by sending commands which the straggler might be missing.

In contrast to the consensus Algorithm 7, the presented SMR algorithm is quite more involved. For clarity, the algorithm is presented in two parts: Algorithm 8 and Algorithm 9. In Algorithm 8, we present the local variables of each replica and the code each replica executes in every round,

⁴Clients need not necessarily operate in lock-step, but can employ pipelining, i.e., can have multiple commands outstanding. Practical systems employ pipelining [2, 3, 100], and we account for this aspect later in our practical experiments of Section 3.4.

⁵We consider that a command is applied instantaneously on the state machine (i.e., execution time for any command is zero).

and in Algorithm 9 we present the two main procedures of the algorithm: `prepareMessages` and `onReceive`.

The high-level overview of the algorithm is that processes decide on a command similar to Algorithm 7 and can help each other by sending commands to processes that are missing them. A bit more specific, each process contains an ordered log of decided commands. For a command to appear in the i -th position of this log, processes need to agree by performing consensus instance i . Processes propose a command for the next consensus instance they are missing and if this position is already decided, other processes will try to help them by sending them their missing commands. In order to be able to help, each process has information⁶ on the next consensus instance each other process tries to decide upon.

Algorithm 8 State Machine Replication: Local Variables for Process p_i and Flow in Each Round

```

1:  $\triangleright$  Local Variables
2:  $ins \leftarrow 1$   $\triangleright$  next consensus instance number to get a decision for
3:  $maxIns \leftarrow 0$   $\triangleright$  greatest consensus instance number where a value is decided upon
4:  $nextMissingIns[p] \leftarrow 1 (\forall p \in \Pi \setminus \{p_i\})$   $\triangleright$  next instance each process needs
5:  $cmdsSet \leftarrow \emptyset$   $\triangleright$  set of commands that are received from the client
6:  $cmdsDecided[i] \leftarrow \perp (\forall i \in \mathbb{N}^+)$   $\triangleright$   $cmdsDecided[i]$  is the command decided for consensus instance  $i$ ,  $\perp$  means no decision is known yet
7:  $messageFor[p] \leftarrow \perp (\forall p \in (\Pi \setminus \{p_i\}) \cup \{client\})$   $\triangleright$  messages to be sent in each round
8:  $SM \leftarrow initSM$   $\triangleright$  initialized state machine to be replicated
9:  $myProposal \leftarrow (p_i, \perp, 0)$   $\triangleright$   $p_i$ 's last proposal in the format of  $(pid, value, instance)$ 
10:  $clientResponses[i] \leftarrow \perp (\forall i \in \mathbb{N}^+)$   $\triangleright$  stores all the responses destined for the client
11:  $lastResponse \leftarrow 1$   $\triangleright$  index of  $clientResponses$ 
12:
13: while new round do
14:    $send(messageFor)$ 
15:    $responses \leftarrow receive()$ 
16:    $onReceive(responses, myProposal)$ 
17:    $(messageFor, myProposal) \leftarrow prepareMessages()$ 

```

We continue by describing the local variables (Algorithm 8). Variable $cmdsDecided$ corresponds to a log of commands⁷ and contains the commands (in order) the process knows have been decided. Note that $cmdsDecided$ allows gaps, for example, a process might have $cmdsDecided[1] \neq \perp$ and $cmdsDecided[3] \neq \perp$ but $cmdsDecided[2] = \perp$. It is guaranteed however that $\forall i \in \mathbb{N}^+ < ins, cmdsDecided[i] \neq \perp$, where ins corresponds to the smallest position in the log the process does not have a command. Similar to ins , $maxIns$ corresponds to the maximum decided instance number of all the other processes. Specifically, $maxIns$ contains the maximum number j such that a process has decided on a command for the j -th position in its log (i.e., there is some process that has $cmdsDecided[j] \neq \perp$). Initially $maxIns$ is zero,

⁶Note that since this information is local, it might become stale and not accurately describe the system.

⁷You can think of the log as the $\ell(e, p)$ construct presented in Section 3.2.

since no process has decided on a command. Each time a process sends a message, it attaches *maxIns* in it, so processes can get informed on the greatest consensus instance number for the whole system where a value has been decided upon. Additionally, each process attaches *ins* to each message it sends so that it can potentially get help from other processes. Helping takes place when a process informs other processes about decided commands they may need. For this, each process has an array (*nextMissing*) of the next instance each process needs. A process looks at this array and sees if it can help another process, and if so it sends the decided command to the other process. Variable *cmdsSet* corresponds to a set of commands that are received from the client and are to be proposed by the process. *messageFor* is set in *prepareMessages* as we will see later on, and it simply contains the message to be sent in every round to the other processes or the client. Additionally, each replica has a state machine *SM* that is initialized to *initSM* and it provides an *apply* operation that takes as a parameter a command and returns the response of applying this command to the state machine. Variable *myProposal* corresponds to a potential command proposed by the process. Finally, the array *clientResponses* is used to store computed responses that are to be sent to the client and *lastResponse* is used to index this array. This array is convenient in case a process is suspended in a round. If this is the case, a process cannot send the response to the client in this round, so the process keeps responses in the *clientResponses* array in order to be able to send a response when it is not suspended.

The exact steps an algorithm executes in a round are presented in lines 13-17: initially the process sends messages, then waits to receive back *responses* that are used together with *myProposal* to change the state of the process and compute the next round's messages (*messageFor*).

We continue by describing how *prepareMessages* and *onReceive* operate. *prepareMessages* operates as follows (Algorithm 9). First, it checks (Line 19) whether it has already decided for instance *ins*. This could happen, if the processes retrieved a decision in Line 60 or in Line 63 of *onReceive*. If the command exists in the set, the command is removed from it (lines 20-21). Afterwards, the command is applied to the state machine (Line 22), the response is stored in *clientResponses* (Line 23) and the latest response that has not yet been transmitted to the client is stored in *messageFor* (Line 24) so it can be sent in the next round to the client. Additionally, *ins* is incremented by one (Line 25). The algorithm then initializes *messageFor* to contain the pair (*ins*, *maxIns*) (Line 26) for every message to be sent to each other process. Including this pair in each message is helpful, since *ins* allows other processes to know the consensus instance the process needs a command for, and *maxIns* ensures that processes only accept proposals for positions that have not yet been decided. Then, *myProposal* (Line 27) is cleared, since otherwise the algorithm might use a previous proposal message. Afterwards, if there exists a command (Line 28) to be proposed and that is not decided yet by the process (Line 30), the process concatenates the proposal to each message (the construct `||` corresponds

Algorithm 9 State Machine Replication (for process p_i)

```

18: procedure prepareMessages()
19:   if  $cmdsDecided[ins] \neq \perp$  then
20:     if  $cmdsDecided[ins] \in cmdsSet$  then
21:        $cmdsSet.remove(cmdsDecided[ins])$ 
22:        $response \leftarrow SM.apply(cmdsDecided[ins])$ 
23:        $clientResponses[ins] \leftarrow response$ 
24:        $messageFor[client] \leftarrow clientResponses[lastResponse]$ 
25:        $ins \leftarrow ins + 1$ 
26:    $messageFor[p] \leftarrow (ins, maxIns), \forall p \in \Pi \setminus \{p_i\}$ 
27:    $myProposal \leftarrow (p_i, \perp, 0) \triangleright$  clear last proposal
28:   if  $cmdsSet \neq \emptyset$  then
29:      $cmd \leftarrow cmdsSet.get()$   $\triangleright$  returns but does not remove an element from the set
30:     if  $\forall j : cmdsDecided[j] \neq cmd$  then  $\triangleright$  not decided yet in which order to execute  $cmd$ 
31:        $\forall p \in \Pi \setminus \{p_i\}, messageFor[p] \leftarrow messageFor[p] \parallel pro(cmd, ins)$ 
32:        $myProposal \leftarrow (p_i, cmd, ins)$ 
33:     else
34:        $cmdsSet.remove(cmd)$   $\triangleright$  already have decided on  $cmd$ , so no need to propose it
35:   for  $p \in \Pi \setminus \{p_i\}$  do
36:     if  $\exists j : nextMissingIns[p] = j \wedge cmdsDecided[j] = cmd \neq \perp$  then
37:        $messageFor[p] \leftarrow messageFor[p] \parallel dec(cmd, nextMissingIns[p])$ 
38:   return ( $messageFor, myProposal$ )
39:
40: procedure onReceive( $responses, myProposal$ )
41:   if  $(client, cmd) \in responses$  then
42:      $cmdsSet.put(cmd)$ 
43:      $responses \leftarrow responses \setminus \{(client, cmd)\}$ 
44:    $\triangleright$  a received message sent by process  $p_j$  is of the format  $p_j, A \parallel B \parallel C$ , where
45:    $\triangleright A = (ins_j, maxIns_j), B = pro(cmd_j, ins_j)$  and  $C = dec(cmd_j, nextMissing_j)$ 
46:   if  $responses \neq \emptyset$  then  $\triangleright$  else,  $p_i$  is suspended in this round
47:     if  $messageFor[client] \neq \perp$  then
48:        $lastResponse \leftarrow lastResponse + 1$ 
49:        $messageFor[client] \leftarrow \perp$ 
50:      $proposals \leftarrow decisions \leftarrow \emptyset$ 
51:     for  $p_j, (ins_j, maxIns_j) \parallel pro(pcmd_j, pins_j) \parallel dec(dcmd_j, nextMiss_j)$  in  $responses$  do
52:        $maxIns \leftarrow \max(maxIns, maxIns_j)$ 
53:        $nextMissing[p_j] \leftarrow ins_j + 1$ 
54:        $proposals \leftarrow proposals \cup \{(p_j, pcmd_j, pins_j)\}$ 
55:        $decisions \leftarrow decisions \cup \{(dcmd_j, nextMiss_j)\}$ 
56:      $proposals \leftarrow proposals \cup \{myProposal\}$ 
57:     if  $\exists (\_, \_, pins) \in proposals : pins = maxIns + 1$  then
58:        $commands \leftarrow \{pcmd : \exists (\_, pcmd, pins) \in proposals : pins = maxIns + 1\}$ 
59:        $decision \leftarrow deterministicFunction(commands)$ 
60:        $cmdsDecided[maxIns + 1] \leftarrow decision$ 
61:        $maxIns \leftarrow maxIns + 1$ 
62:     for  $\forall (dmcd, nextMissing) \in decisions : cmdsDecided[nextMissing] = \perp$  do
63:        $cmdsDecided[nextMissing] \leftarrow dmcd$ 

```

to concatenation of messages) (Line 31), and sets *myProposal* (Line 32). Then, in lines 35 to 37, the process checks if it can help other processes by sending it decided commands it knows that other processes are missing. At the end it returns *messageFor* together with the proposal (Line 38). Finally, note that each message sent to another processes consists of at most three parts, a pair of instance numbers (*ins*, *maxIns*), a propose, and a decided message. The careful reader might notice that consensus instance numbers can grow infinitely large. Hence, messages can potentially have unbounded size. One option to avoid this, is to split a large message into multiple smaller ones and keep the exact same algorithm, with the slight change that it only considers a message when it has accepted all of its smaller messages. Another option is to explicitly state that messages consist of two parts, a header part that contains information such as instance numbers, signatures, etc., and an application part. Then, we can just ask to bound the application part of the messages, but not the header part [39]. We are not concerned with the header, which can grow so as to permit increasingly larger consensus instance numbers. On the other hand, this bound on the application part of a message is important to prevent “cheating” in the sense of batching all the commands from multiple consensus instances in a single message.

onReceive operates as follows. First, it checks whether the client sent a message (Line 41), and if so adds the sent command to *cmdsSet* (Line 42) and removes it from *responses* (Line 43). Then, the process checks if it is suspended in this round (Line 46), and if this is the case it does not perform any other operation. If the process is correct and *messageFor[client] ≠ ⊥* (Line 47) it means that it successfully sent the previous response to the client, so it increases *lastResponse* (Line 48) and clears *messageFor[client]* (Line 49). Then, it initializes *proposals* and *decisions* to be empty sets (Line 50). Then, it goes through the *responses* (Line 51) to update *maxIns* (Line 52), update *nextMissing* for each process (Line 53) and gather proposals and decisions from the responses (lines 54-55)⁸. Afterwards, the process adds its own proposal to the set of proposals (Line 56). Then, if there are proposals (Line 57) for instance number equal to *maxIns* + 1, all the commands are extracted from such proposals (Line 58) (note that we employ pattern matching by using the symbol $_$). The extracted commands are passed through some deterministic function (similar to Algorithm 7) and a value is decided upon (Line 60). Subsequently, *maxIns* is incremented appropriately (Line 61). At the end, the process goes through the decided messages (Line 62) and utilizes delivered commands that it needs.

Finally, note that the algorithm accepts optimizations (e.g., updating the *nextMissing* array after deciding on a command in Line 60), but we omitted them for clarity.

Proof. In what follows we prove that the algorithm satisfies safety (i.e., it is a valid SMR algorithm) and liveness (i.e., a clients eventually gets a response for any command it proposes) in theorems 2 and 3 respectively. We start by proving some useful lemmas. Note that in what

⁸Note that a response might not contain a proposal or a decision.

follows we consider that the commands proposed by the client are always distinguishable. Furthermore, we denote with $variable_p[i]$ the value of $variable[i]$ of process p . Finally, when we state that a variable has a specific value in some round r , we refer to the beginning of round r (exactly before Line 14).

Lemma 12. *For any $p \in \Pi$, $maxIns_p$ never decreases.*

Proof. Note that $maxIns_p$ is only modified in lines 52 and 61. In Line 52, $maxIns_p$ cannot decrease since it is updated to be the maximum of its own value and the $maxIns$ received by some other process, so it will be greater or equal to what it was before. In Line 61, $maxIns_p$ is incremented by one, so again it does not decrease. \square

Lemma 13. *For any $p \in \Pi$ and $i \in \mathbb{N}^+$, if $i \geq maxIns_p + 1$, then $cmdsDecided_p[i] = \perp$.*

Proof. We use induction to prove this lemma. At the beginning of the first round, the property trivially holds. Assume it holds at the beginning of round r . We will show it holds at the beginning of round $r + 1$. During the execution of round r there are two possibilities for $cmdsDecided_p$ to be modified so that the property will not hold. One is for $cmdsDecided_p$ to be written in Line 60 and another to be written in Line 63. If a write occurs in Line 60 the property does not hold but $maxIns_p$ is incremented immediately afterwards in Line 61, so the property holds back up again. The other case is that $cmdsDecided_p$ is written in Line 63. If $nextMissing = maxIns_p + 1$, then the property will not hold in round $r + 1$. However, this would imply that some other replica sent a decided message with $maxIns_p + 1$ in the previous round. But then $maxIns_p$ in Line 52 would have been updated to correspond to this fact, a contradiction. Finally, note that due to Lemma 12 $maxIns_p$ never decreases, so the property cannot be circumvented by a reduction in $maxIns_p$. Therefore, the property holds at the beginning round $r + 1$ and hence of every round. \square

Lemma 14. *For any $p \in \Pi$ and $i \in \mathbb{N}^+$, $cmdsDecided_p[i]$ is written at most once.*

Proof. We note that $cmdsDecided_p[i]$ for a specific $i \in \mathbb{N}^+$ is only updated in Algorithm 9 and in two places: Line 60 and Line 63. Furthermore, updates to $cmdsDecided_p$ take only place by processes that were not suspended in this round (Line 46). An update of $cmdsDecided_p$ in Line 60 occurs for the index $maxIns_p + 1$. Note that $maxIns_p$ since the start of the round might have only increased in Line 52, so the result of Lemma 13 is still satisfied. Due to Lemma 13, $cmdsDecided_p[maxIns_p + 1]$ will be \perp . Hence updates in Line 60 can only occur in slots of $cmdsDecided_p$ that do not contain a command (i.e., a slot i such that $cmdsDecided_p[i] = \perp$). Therefore, slots that contain commands will not get overwritten in Line 60. Similarly, the update in Line 63 only occurs if $cmdsDecided_p[i] = \perp$ (where $i = nextMissing$) and not otherwise,

so a $cmdsDecided_p[i]$ that already contains a command ($\neq \perp$) will not get updated in Line 63. Therefore, a specific position in $cmdsDecided_p$ gets updated at most once. \square

Lemma 15. *For any two processes $p, p' \in \Pi$ that are correct in a round r , then $maxIns_p = maxIns_{p'}$ immediately after Line 52 in round r .*

Proof. For this lemma, we use a similar argument to the one used to prove that the consensus Algorithm 7 satisfies agreement. All the correct processes would be able to deliver their local $maxIns$. Then each correct process will apply get the maximum for all the $maxIns$ values it delivered (Line 52), as well as its own. Hence, they will have the exact same value. \square

Theorem 2. *Algorithm 9 is a valid SMR algorithm.*

Proof. To show that the algorithm is valid, we have to show that the logs ($cmdsDecided$) are always consistent with each other.

We prove by induction that Algorithm 9 has the following property: for any two processes $p, p' \in \Pi$, if $cmdsDecided_p[i] \neq cmdsDecided_{p'}[i]$ for some $i \in \mathbb{N}^+$, then $cmdsDecided_p[i] = \perp$ or $cmdsDecided_{p'}[i] = \perp$. In the first round, the property trivially holds since $cmdsDecided_p[i] = \perp \forall i \in \mathbb{N}^+$ and $\forall p \in \Pi$. Assume the property holds for all the rounds up to the r -th one. We will prove that it holds for round $r + 1$. For this, we note that $cmdsDecided$ is updated in two different places, so we consider two cases:

- If a proposal takes place in Line 60. Due to Lemma 15, all correct processes will have the exact same $maxIns$ values, so they would all consider the exact same set of proposals (Line 58). Similar to the consensus Algorithm 7, correct processes will choose the exact same set of commands as well and pass them through the `deterministicFunction` to make a decision. Therefore, all replicas will store the exact same *decision* in their $cmdsDecided$ log and hence the property still holds.
- If $cmdsDecided$ is updated in Line 63, this means that the process received a decided message. This decided message was computed in a previous round (Line 37), but in the previous round the property holds (by induction). Therefore, if more than one process updates the *nextMissing* array, they will update it with the exact same command.

Finally, since the same log position is never updated more than once due to Lemma 14, the property will always be satisfied. Therefore, based on the definition of a valid SMR Algorithm (Section 3.2), Algorithm 3 is valid. \square

Lemma 16. *If $cmdsDecided_p[i] = cmd \neq \perp$ for some process $p \in \Pi$, then there are at least $n - 1$ processes with $cmdsDecided[i] = cmd$.*

Proof. In other words, this lemma states that each decided command exists in the log of $n - 1$ replicas. When a command is first decided upon in Line 60, every correct process at that round (and there are at least $n - 1$ correct processes in each round) decides on the exactly same command, since all correct processes see the exact same *maxIns* value (Lemma 15). Hence, at least $n - 1$ processes will store this command in *cmdsDecided*. \square

Lemma 17. *If $\text{cmdsDecided}_p[i] \neq \perp$, then eventually at least $n - 1$ processes, for all $j : 1 \leq j \leq i$ will have $\text{cmdsDecided}[j] \neq \perp$.*

Proof. Recall, that in every round, every process sends *ins* (Line 26) to the other processes informing them on the instance it is currently trying to get a command for. If the process is correct in the round, at least $n - 1$ of the other correct processes in this round will deliver the message and update their local *nextMissing* array (Line 53). From Lemma 16 we know that each decided command exists in the log of at least $n - 1$ processes. Since $n \geq 3$ and at most one process can be suspended in a round, we know that there will always be one correct process that contains a command that we are missing and that can help (Line 37). Therefore, in the next round if this process is correct it will receive a decision. Hence, eventually at least $n - 1$ processes will eventually fill up their log up to position i . \square

Lemma 18. *If a command c is decided ($\exists p \in \Pi$ and $\exists j \in \mathbb{N}^+$ such that $\text{cmdsDecided}_p[j] = c$), then the client eventually gets a response for command c .*

Proof. Due to Lemma 17, we know that if a command c appears in the log of some process, eventually c will appear in the logs of at least $n - 1$ processes, as well as all the previous commands in the log. Each process would have applied the command to the local state machine and stored the response in *clientResponses* (Line 23). Note however, that a process sends a response of a command to the client only if it is certain that the response to the previous command has been successfully delivered by the client. So only if the process is correct in a round and it can be assured that the previous response message was sent to the client (see lines 48 and 49), only then, the process sends the response to the next command to the client. Therefore, the client will eventually get a response for command c . \square

Theorem 3. *If a client suggests a command, then the client eventually gets a response.*

Proof. Assume by contradiction that the k -th command c_k is the first command that is suggested by the client in which the client never gets a response for. We will prove that the client will eventually get a response for c_k and hence a contradiction. First, note that since at least $n - 1$ processes are correct in each round, at least $n - 1$ processes will add c_k to their log (Line 42). The client only suggests a command if it received a response for the previous command (see Section 3.2). Hence, processes have already decided on the c_{k-1} command.

From Lemma 16, we know that c_{k-1} exists in the log of at least $n - 1$ processes. Furthermore, due to Lemma 17, eventually all processes will fill their logs up to command c_{k-1} . This means that eventually $cmdsSet$ will contain c_k as the first command for at least $n - 1$ processes, since all the other commands will be decided and removed from the set (Line 21 and 34). Therefore, c_k will be eventually proposed with instance number k , where $k = maxIns_p + 1$ for some process p , and hence it will be decided. Finally, from Lemma 18, since command c_k is decided, the client will eventually get a response for this command. \square

As we show next (Section 3.3), no SMR algorithm can respond to a client in a finite number of rounds. Hence, even with helping, our SMR algorithm cannot guarantee a constant response either.

3.3 Complexity Lower Bound on State Machine Replication

We now present the main result of this chapter, namely we show that there is no State Machine Replication (SMR) algorithm that can always respond to a client in a constant number of rounds. We also discuss how this result extends beyond the model of Section 3.2.

3.3.1 Complexity Lower Bound

We briefly describe the idea behind our result. We observe that there is a bounded number of commands that can be delivered by a replica in a single round, since messages are of bounded size, a practical assumption (Lemma 19). Using this observation, we show that in a finite execution e , if each replica p_i is missing β_i commands, then an SMR algorithm needs $\Omega(\min_i \beta_i)$ rounds to respond to at least one client command suggested in $e^+ \in E(e)$ (Lemma 20). Finally, for any $r \in \mathbb{N}^+$, we show how to construct an execution e where each replica misses enough commands in e , so that a command suggested by a client in $e^+ \in E(e)$ cannot get a response in less than r rounds (Theorem 4). Hence, no SMR algorithm in our model can respond to every client command in a constant number of rounds.

Lemma 19. *A single replica can deliver up to a bounded number (that we denote by Ψ) of commands in a round.*

Proof. Since any message m is of bounded size B ($\forall m \in M, |m| \leq B$), the number of commands message m can contain is bounded. Let us denote with ψ the maximum number of commands any message can contain. Since the number of commands that can be contained in one message is at most ψ , a replica can transmit at most ψ commands to another replica in one round. Therefore, in a given round a replica can deliver from other replicas up to $\Psi = (n - 1)\psi$ commands. In other words, a replica cannot recover faster than Ψ commands per round. \square

3.3. Complexity Lower Bound on State Machine Replication

Lemma 20. *For any finite execution e , if each replica p_i misses β_i commands (i.e., p_i is a β_i -straggler), then there is a command suggested by the client in some execution $e^+ \in E(e)$ such that we need at least $\lceil \min_i (\beta_i / \Psi) \rceil$ rounds to respond to it.*

Proof. Consider an execution $e^+ \in E(e)$ such that in a given round r , a client suggests to all replicas a command c , where round r exists in e^+ but does not exist in e . This implies that replicas are not yet aware of command c in e , so command c should appear in a log position i where i is greater than $\max_p |\ell(e, p)|$. In order for a replica to respond to the client's command c , the replica first needs to have all the commands preceding c in its log. For this to happen, some replica needs to get informed about β_i commands. Note that from Lemma 19, a replica can only deliver Ψ commands in a round. Therefore, a replica needs at least $\lceil \beta_i / \Psi \rceil$ rounds to get informed about the commands it is missing (i.e., recover), and hence we need at least $\lceil \min_i (\beta_i / \Psi) \rceil$ rounds for the client to get a response for c . \square

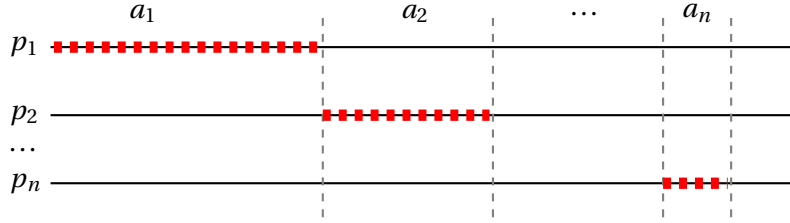


Figure 3.1 – Constructed execution of Theorem 4. Red dashed lines correspond to rounds where a replica is suspended. Replica p_1 is suspended for a_1 rounds, replica p_2 for a_2 rounds, etc.

Theorem 4. *For any $r \in \mathbb{N}^+$ and any SMR algorithm with n replicas ($n \geq 3$), there exists an execution e , such that a command c which the client suggests in some execution $e^+ \in E(e)$ cannot get a response in less than r rounds.*

Proof. Assume by contradiction that, given an SMR algorithm, each command suggested by a client needs at most a constant number of rounds k to get a response. Since we can get a response to a command in at most k rounds, we can make a replica “miss” any number of commands by simply suspending it for an adequate amount of rounds.

To better convey the proof we introduce the notion of a phase. A phase is a conceptual construct that corresponds to a number of contiguous rounds in which a specific replica is suspended. Specifically, we construct an execution e consisting of n phases. Figure 3.1 conveys the intuition behind this execution. In the i -th phase, replica p_i is suspended for a_i rounds, and $a_i \neq a_j$ for $i \neq j$. The idea is that after the n -th phase, each replica is a straggler and needs more than k rounds to become a non-straggler and be able to respond to a client command

suggested in a round o , where o exists in e^+ but not in e . We start from the n -th phase, going backwards. In the n -th phase, we make replica p_n miss enough commands, say β_n . In general, the number β_n of commands is such, that if a client suggests a command at the end of the n -th phase, the client cannot get a response from within k rounds of the command being suggested. For this to hold, it suffices to miss $\beta_n = k\Psi + 1$ commands. In order to miss β_n commands, we have to suspend p_n for at least $\beta_n k$ rounds, since a client may submit a new command every (at most) k rounds. Thus, we set $\alpha_n = \beta_n k$. Similarly, replica p_{n-1} has to miss enough commands (β_{n-1}) such that it cannot get all the commands in less than k rounds. Note that after p_{n-1} was suspended for α_{n-1} rounds, replica p_n took part in α_n rounds. During these α_n rounds, replica p_{n-1} could have recovered commands it was missing. Therefore, p_{n-1} must miss at least $\beta_{n-1} = (\alpha_n + k)\Psi + 1$ commands and $\alpha_{n-1} = \beta_{n-1} k$. In the same vein, $\forall i \in \{1, \dots, n\} \beta_i = ((\sum_{j=i+1}^n \alpha_j) + k)\Psi + 1$.

With our construction we succeed in having $\beta_i/\Psi = (\sum_{j=i+1}^n \alpha_j) + k + 1/\Psi > k$ for every $i \in \{1, \dots, n\}$. Therefore, using Lemma 20, after the n phases, each replica needs more than k rounds to get informed about commands it is missing from its log, a contradiction. \square

Theorem 4 states that there exists no SMR algorithm in our model that can respond to every client command in a constant number of rounds.

3.3.2 Extension to other Models

The system model we use (Section 3.2) lends itself to capture naturally the difference in complexity (i.e., number of rounds) between consensus and SMR. It is natural to ask whether this difference extends to other system models—and which are those models. Identifying all the models where our result applies, or does not apply, is a very interesting topic which is beyond the scope of this work, but we briefly discuss it here.

Consider models which are stronger than ours. An example of a stronger model is one that is synchronous with no failures; such a model would disallow stragglers and hence both consensus and SMR can be solved in constant time. Similarly, if the model does not restrict the size of messages (see Lemma 19), then an SMR command can complete in constant time, circumventing our result. We further discuss how our result can be circumvented in Section 3.5.

A more important case is that of weaker, perhaps more realistic models. If the system model is too weak—if consensus is not solvable [51]—then it is not obvious how consensus relates to SMR in terms of complexity. Such a weak model, however, can be augmented, for instance with unreliable failure detectors [37], allowing consensus to be solved. Informally, during well-

behaved executions of such models, i.e., executions when the system behaves synchronously and no failures occur [68], SMR commands can complete in constant time.

Most practical SMR systems [36, 40, 96, 100] typically assume a partially synchronous or an asynchronous model with failure detectors [37], and executions are not well-behaved, because failures are prone to occur [19]. We believe our result applies in these practical settings, concretely within synchronous periods (or when the failure detector is accurate, respectively) of these models. During such periods, if at least one replica can suffer message omissions, completing an SMR command can take a non-constant amount of time. Indeed, in the next section, we present an experimental evaluation showing that our result holds in a partially synchronous system.

3.4 The Empirical Perspective

Our goal in this section is to substantiate empirically the theoretical result of Section 3.3. We first cover details of the experimental methodology. Then we discuss the evaluation results both in a single-machine environment, as well as on a practical wide-area network (WAN).

3.4.1 Experimental Methodology

We use two well-known State Machine Replication (SMR) systems: (1) LibPaxos, a Multi-Paxos implementation [3], and (2) etcd [2], a mature implementation of the Raft protocol [100]. We note that LibPaxos distinguishes between three roles of a process: proposer, acceptor, and learner [78]. To simplify our presentation, we unify the terminology so that we use the term *replica* instead of *acceptor*, the term *client* replaces *learner*, and the term *leader* replaces *proposer*. Each system we deploy consists of three replicas, since this is sufficient to validate our result and moreover it is a common deployment practice [40, 55]. We employ one client. In LibPaxos, we use a single leader, which corresponds to a separate role from replicas. In Raft, one of the three replicas acts as the leader.

Using these two systems, we measure how consensus relates to SMR in terms of cost in the following three scenarios:

1. **Graceful:** when network conditions are uniform and no failures occur; this scenario only applies to the single-machine experiments of Section 3.4.2;
2. **Straggler:** a single replica is slower than the others (i.e., this is a straggler) but no failures occur, so the SMR algorithm needs not rely on the straggler;
3. **Switch:** a single replica is a straggler and a failure occurs, so the SMR algorithm has to

include the straggler on the critical path of agreement on commands.

Due to the difficulty of running synchronous rounds in a practical system, our measurements are not in terms of rounds (as in the model of Section 3.2). Instead, we take a lower-level perspective. We report on the *cost*, i.e., number of messages, and the *latency* measured at the client.⁹ Specifically, in each experiment, we report on the following three measurements.

First, we present the cost of each consensus instance i in terms of number of messages which belong to instance i , and which were exchanged between replicas, as well as the client. Each consensus instance has an identifier (called *iid* in LibPaxos and *index* in Raft), and we count these messages up to the point where the instance completes at the client. Recall that in our model (Section 3.2.1) we similarly consider consensus to complete when the client learns the decided value. This helps us provide an “apples to apples” comparison between the cost of consensus instances and SMR commands (which we describe next).

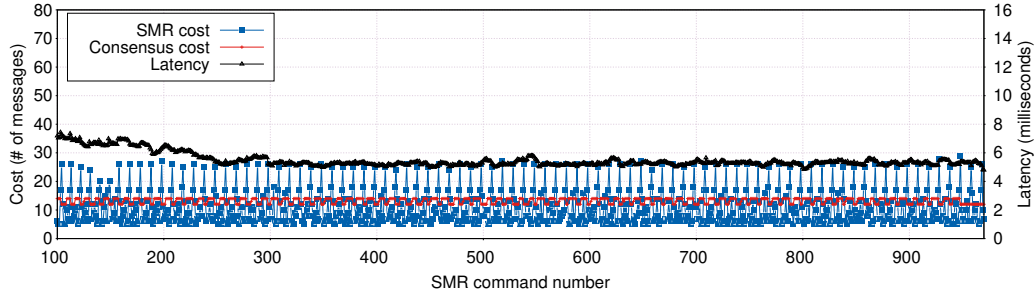
Second, we measure the cost of each SMR command c . Each command c is associated with a consensus instance i . The cost of c is similar to the cost of i : we count messages exchanged between replicas and the client for instance i .¹⁰ The cost of a command c , however, is a more nuanced measurement. As we discussed already, a consensus instance typically leaks messages, which can be processed later. Also, both systems we consider use pipelining, so that a consensus instance i may overlap with other instances while a replica is working on command c . Specifically, the cost of c can include messages leaked from some instance j , where $j < i$ (because a replica cannot complete command c without having finished all previous instances) but also from some instance k , with $k > i$ (these future instances are being prepared in advance in a pipeline, and are not necessary for completing command c).

Third, we measure the latency for completing each SMR command. An SMR command starts when the client submits this command to the leader, and ends when the client learns the command. In LibPaxos, this happens when the client gathers replies for that command from two out of three replicas; in Raft, the leader notifies the client with a response.

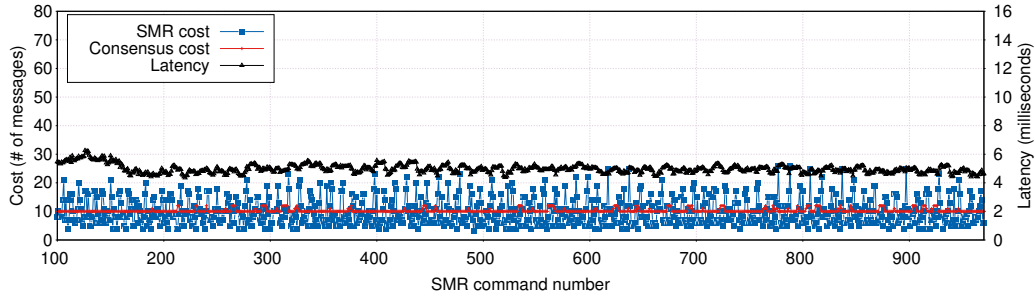
We consider both a single-machine setup and a WAN. The former setup serves as a controlled environment where we can vary specifically the variable we seek to study, namely the impact of a straggler when quorums switch. For this experiment, we use LibPaxos and we discuss the results thoroughly. The latter setup reflects real-world conditions which we use to validate against our findings in the single-machine setup, and we experiment with both systems. In all executions the client submits 1000 SMR commands; we ignore the first 100 (warm-up) and the last 50 commands (cool-down) from the results. We run the same experiment three times to confirm that we are not presenting outlying results.

⁹Note that it is simple to convert rounds to messages, considering our description of rounds in Section 3.2.

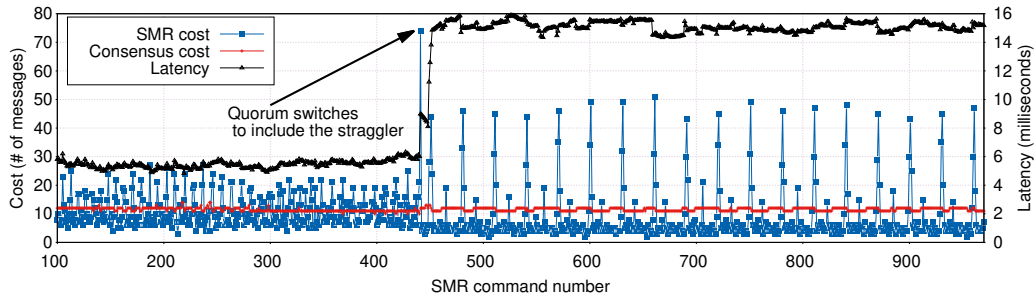
¹⁰For LibPaxos, the cost of consensus and SMR includes additionally messages involving the leader.



(a) **Graceful** scenario: all replicas experience uniform conditions and no failures occur.



(b) **Straggler** scenario: one of the three replicas is a straggler.



(c) **Switch** scenario: one of the three replicas is a straggler and the active quorum switches to include this straggler.

Figure 3.2 – Experimental results with LibPaxos on a single-machine setup. We compare the cost of SMR commands with the cost of consensus instances in three scenarios.

3.4.2 Experimental Results on a Single Machine

We experiment on an Intel Core i7-3770K (3.50GHz) equipped with 16GB of RAM. Since there is no network in these experiments, spurious network conditions—which can arise in practice, as we shall see next in Section 3.4.3—do not create noise in our results. To make one of the replicas a straggler, we make this replica relatively slower through a random delay (via the `select` system call) of up to 500 μ s when this replica processes a protocol message.

In Figure 3.2a we show the evolution of the three measurements we study for the **graceful** execution. The mean latency is 5590 us with a standard deviation of 730 us, i.e., the performance is very stable. This execution serves as a baseline.

In Figure 3.2b we present the result for the **straggler** scenario. The average latency, compared with Figure 3.2a, is slightly smaller, at 5005 us; the standard deviation is 403 us. The explanation for this decrease is that there is less contention (because the straggler backs-off periodically), so the performance increases. In this scenario, additionally, there is more variability in the cost of SMR commands, which is a result of the straggler replica being less predictable in how many protocol messages it handles per unit of time.

For both Figures 3.2a and 3.2b, the average cost of an SMR command is the same as the average cost of a consensus instance, specifically around 12 messages. There is, however, a greater variability in the cost of SMR commands—ranging from 5 to 30 messages—while consensus instances are more regular—between 11 and 13 messages. As we mentioned already, the variability in the cost of SMR springs from two sources: (1) consensus instances leak into each other, and (2) the use of pipelining, a crucial part in any practical SMR algorithm, which allows consensus instances to overlap in time [64, 107].

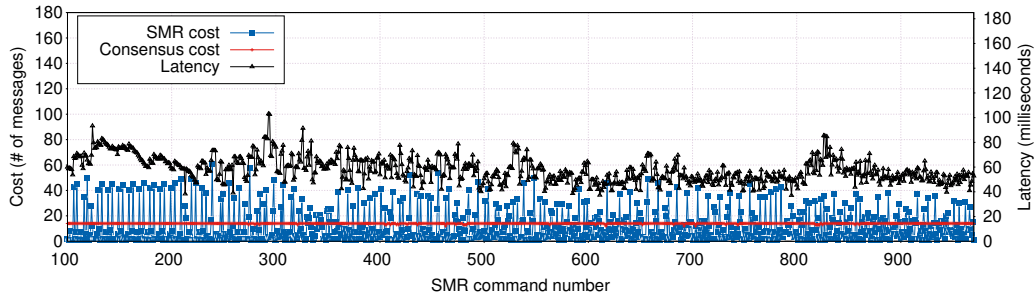
Pipelining allows the leader to have multiple outstanding proposals, and these are typically sent and delivered in a burst, in a single network-level packet. This means that some commands can comprise just a few messages (all the other messages for such a command have been processed earlier with previous commands, or have been deferred), whereas some commands comprise many more messages (e.g., messages leaked from previous commands, or processed in advance from upcoming commands). In our case, the pipeline has size 10, and we can distinguish in the plots that the bumps in the SMR cost have this frequency. Larger pipelines allow higher variability in the cost of SMR. Importantly, to reduce the effect of pipelining on the cost of SMR commands, this pipeline size of 10 is much smaller than it is used in practice, which can be 64, 128, or larger [2, 3].

Figure 3.2c shows the execution where we stop one replica, so the straggler has to take part in the active quorum. The moment when the straggler has to recover all the missing state and start participating is evident in the plot. This happens at SMR command 450. We observe that SMR command 451 has considerably higher cost. This cost comprises all the messages which the straggler requires to catch-up, before being able to participate in the next consensus instance. The cost of consensus instance 451 itself is no different than other consensus instances. Since the straggler becomes the bottleneck, the latency increases and remains elevated for the rest of the execution. The average latency in this case is noticeably higher than in the two previous executions, at 10730 us (standard deviation of 4726 us). For this execution, we observe the same periodical bumps in the cost of SMR commands. Because the straggler replica is on the critical path of agreement, these bumps are more pronounced and less frequent: the messages

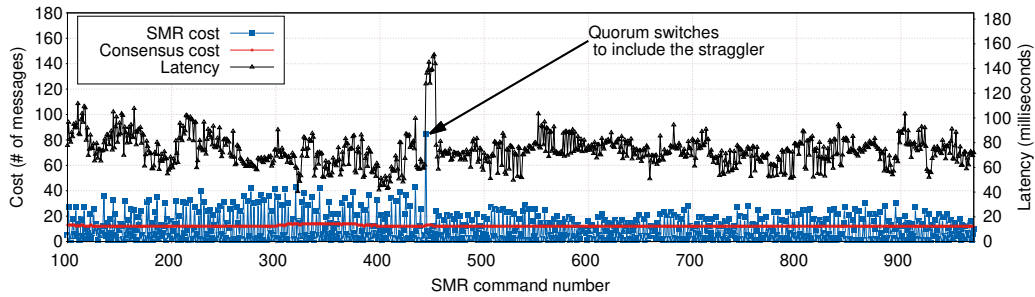
concerning the straggler (including to and from other replicas or the client) accumulate in the incoming and outgoing queues and are processed in bursts.

3.4.3 Wide-area Experiments

We deploy both LibPaxos and Raft on Amazon EC2 using *t2.micro* virtual machines [1]. For LibPaxos, we colocate the leader with the client in Ireland, and we place the three replicas in London, Paris, and Frankfurt, respectively. Similarly, for Raft we colocate the leader replica along with the client in Ireland, and we place the other two replicas in London and Frankfurt. Under these deployment conditions, the replica in Frankfurt is naturally the straggler, since this is the farthest node from Ireland (where the leader is in both systems). Therefore, we do not impose any delays, as we did in the earlier single-machine experiments. Furthermore, colocating the client with the leader minimizes the latency between these two, so the latency measurements we report indicate the actual latency of SMR.



(a) **Straggler** scenario: the replica in Frankfurt is a straggler, since this is the farthest from the leader in Ireland. The system forms a quorum using the replicas in London and Paris.

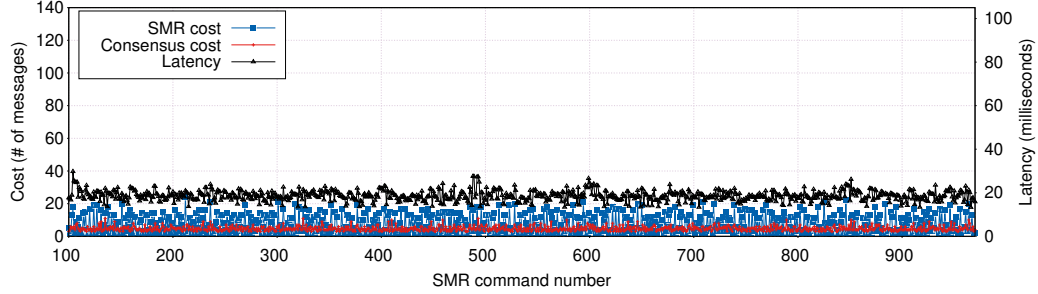


(b) **Switch** scenario: at SMR command 450 we switch out the replica in London. The straggler in Frankfurt then becomes part of the active quorum.

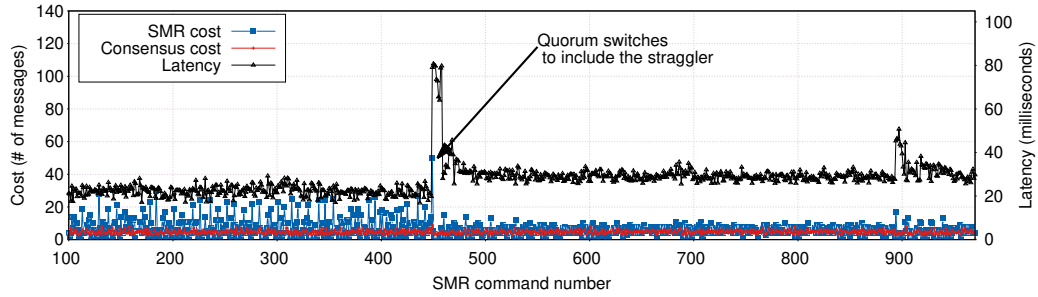
Figure 3.3 – Experimental results with LibPaxos on the WAN. Similar to Figure 3.2, we compare the cost of SMR commands with the cost of consensus instances.

Figures 3.3 and 3.4 present our results for LibPaxos and Raft, respectively. To enhance visibility,

please note that we use different scales for the y and y_2 axes. These experiments do not include the **graceful** scenario, because the WAN is inherently heterogeneous.



(a) **Straggler** scenario: the replica in Frankfurt is a straggler. The active quorum consists of the leader in Ireland and the replica in London.



(b) **Switch** scenario: we stop the replica in London at SMR command 450. Thereafter, the active quorum must switch to include the straggler in Frankfurt.

Figure 3.4 – Experimental results with Raft on the WAN. Similar to Figures 3.2 and 3.3, we compare the cost of SMR commands with the cost of consensus instances.

The most interesting observation is for the **switch** scenarios, i.e., Figures 3.3b and 3.4b. In these experiments, when we stop one of the replicas at command 450, there is a clear spike in the cost of SMR, which is similar to the spike in Figure 3.2c. Additionally, however, there is also a spike in latency. This latency spike does not manifest in single-machine experiments, where communication delays are negligible. Moreover, on the WAN the latency spike extends over multiple commands, because the system has a pipeline so the latency of each command being processed in the pipeline is affected while the straggler is catching up. After this spike, the latency decreases but remains slightly more elevated than prior to the switch, because the active quorum now includes the replica from Frankfurt, which is slightly farther away; the difference in latency is roughly 5 ms.

Beside the latency spike at SMR command 450, these experiments reveal a few other glitches, for instance around command 830 in Figure 3.3a, or command 900 in Figure 3.4b. In fact, we observe that unlike our single-machine experiments, the latency exhibits a greater variability.

As we mentioned already, this has been observed before [34, 96, 123] and is largely due to the heterogeneity in the network and the spurious behavior this incurs. This effect is more notable in LibPaxos, but Raft also shows some variability. The latter system reports consistently lower latencies because an SMR command completes after a single round-trip between the leader and replicas [100].

As a final remark, our choice of parameters is conservative, e.g., execution length or pipeline width. For instance, in executions longer than 1000 commands we can exacerbate the difference in cost between SMR commands and consensus instances. Longer executions allow a straggler to miss even more state which it needs to recover when switching.

3.5 Discussion

The main implication of Theorem 4 is that it is impossible to devise a State Machine Replication (SMR) algorithm that can bound its response times. There are several conditions, however, which allow to circumvent our lower bound, which we discuss here. Moreover, when our result does apply, we observe that SMR algorithms can mitigate, to some degree, the performance degradation in the worst-case, i.e., when quorums switch and stragglers become necessary. These algorithms experience a trade-off between best-case and worst-case performance. We also discuss how various SMR algorithms deal with this trade-off.

Circumventing the Lower Bound. Informally, our result applies to SMR systems which fulfill two basic characteristics: i) messages are bounded in size, and ii) replicas can straggle for arbitrary lengths of time. Simply put, if one of these conditions does not hold, then we can circumvent Theorem 4. We discuss several cases when this can happen.¹¹

For instance, if the total size of the state machine is bounded, as well as small in size, then the whole state machine can potentially fit in a single message, so a straggler can recover in bounded time. This is applicable in limited practical situations. We are not aware of any SMR protocol that caps its state. But this state can be very small in some applications, e.g. if SMR is employed towards replicating only a critical part of the application, such as distributed locks or coordination kernels [64, 89].

The techniques of load shedding or backpressure [122] can be employed to circumvent our result. These are application-specific techniques which, concretely, allow a system to simply drop or deny a client command if the system cannot fulfill that command within bounded time. Other, more drastic, approaches to enforce strict latencies involve resorting to weak consistency or combining multiple consistency models in the same application [59], or provisioning

¹¹We do not argue that we can guarantee bounded response times in a general setting, only in the model we consider in Section 3.2.

additional replicas proactively when stragglers manifest [42, 111].

Best-case Versus Worst-case Performance Trade-off. When our lower bound holds, an SMR algorithm can take steps to ameliorate the impact which stragglers have on performance in the worst-case (i.e., when quorums switch). Coping with stragglers, however, does not come for free. The best-case performance can suffer if this algorithm expends resources (e.g., additional messages) to assist stragglers. Concretely, these resources could have been used to sustain a higher best-case throughput. When a straggler becomes necessary in an active quorum, however, this algorithm will suffer a smaller penalty for switching quorums and hence the performance in the worst-case will be more predictable.

This is the trade-off between best- and worst-case performance, which can inform the design of SMR algorithms. Most of the current well-known SMR protocols aim to achieve superior best-case throughput by sacrificing worst-case performance. This is done by reducing the replication factor, also known as a *thrifty* optimization [96]. In this optimization, the SMR system uses only $F + 1$ instead of $2F + 1$ replicas—thereby stragglers are non-existent—so as to reduce the amount of transmitted messages and hence improve throughput or other metrics [3, 85, 96]. In the worst-case, however, when a fault occurs, this optimization requires the SMR system to either reconfigure or provision an additional replica on the spot [84, 85], impairing performance.

Multi-Paxos proposes a mode of operation that can strike a good balance between best- and worst-case performance [77]. Namely, replicas in this algorithm can have gaps in their logs. When gaps are allowed, a replica can participate in the agreement for some command on log position k even if this replica does not have earlier commands, i.e., commands in log positions l with $l < k$. As long as the leader has the full log, the system can progress. Even when quorums switch, stragglers can participate without recovery. If the leader fails, however, the protocol halts [92, 120] because no replica has the full log, and execution can only resume after some replica builds the full log by coordinating with the others. It would be interesting in future work to experiment with an implementation that allows gaps, but LibPaxos does not follow this approach [3], and we are not aware of any such implementation.

3.6 Conclusion

We examined the relation between consensus and State Machine Replication (SMR) in terms of their complexity. We proved the surprising result that SMR is more expensive than a repetition of consensus instances. Concretely, we showed that in a synchronous system where a single instance of consensus always terminates in a constant number of rounds, completing one SMR command can potentially require a non-constant number of rounds. Such a scenario can

occur if some processes are stragglers in the SMR algorithm, but later the stragglers become active and are necessary to complete a command. We showed that such a scenario can occur if even one process is a straggler at a time.

Additionally, we supported our formal proof with experimental results using two well-known SMR implementations (a Multi-Paxos and a Raft implementation). Our experiments highlighted the difference in cost between a single consensus instance and an SMR command.

Transactions **Part II**

4 The Impossibility of Fast Transactions

In this chapter, we prove that transactions cannot be fast in an asynchronous fault-tolerant system. Our result holds in any system where we require transactions to ensure monotonic writes, or any stronger consistency model, such as, causal consistency. Thus, our result unveils an important, and so far unknown, limitation of fast transactions: they are impossible if we want to tolerate the failure of even one server.

4.1 Introduction

The surge of cloud computing and big data has led to the design of large-scale and highly available online services. A fundamental component of large-scale online services is a distributed data store [4, 43, 74]. Naturally, the demand for highly available online services translates to the demand for highly available data stores.

The CAP theorem [27, 56] states that a distributed system has to choose between availability and strong consistency during a network partition. In practice, networks are not reliable [19], and thus we can never eliminate the possibility of network partitions. For this reason, highly available data stores choose to sacrifice strong consistency in favor of availability [4, 43, 70, 74]. A number of well-known data stores support eventual consistency [4, 43, 70]. Eventual consistency [65] simply states that if we reach a quiescent state where no updates are taking place (i.e., no writes are issued by the clients), eventually all the servers contain non-conflicting data.¹ Recent work has shown that the strongest consistency model that can be achieved in the presence of network partitions is causal consistency [14, 90]. As a result, causal consistency has recently gained attention in academia [18, 45, 86, 87, 112–114], as well as in industry, where most notably, the MongoDB [4] data store supports causal consistency.

¹In contrast to its name, eventual consistency is a liveness, rather than a safety property.

Typically, the interface of a data store is a read-write interface [74] on a set of objects, where the objects can be identified by what is called a key. To handle the enormous amount of data, data stores partition² the data (e.g., based on a key) across multiple servers. To avoid loss of data, these systems replicate the data to multiple servers. To remain highly available and reduce the latency of client operations, data stores are replicated across multiple geographically separated data centers. At a high-level, the client issues a request on the data store by identifying the object (e.g., by providing the key) the client wants to access. Subsequently, the server responds back to the client with the desired data. A number of data stores [4, 9, 45, 47, 86, 87, 94] augment their interface by providing transactions. Transactions operate on multiple objects at once and substantially aid the programmer's job. Data stores being extensively used for read-heavy workloads, aim to optimize read-only transactions since they are the most frequent in practice [28]. It is thus natural to seek implementations for read-only transactions that are as fast as possible.

Lu et al. [87] provide a first informal description of what it means for a read-only transaction to be *fast*, or as they call it, *latency-optimal*. Their definition, captures the fact that a read-only transaction is one round-trip, non-blocking, and one-version. One round-trip means that a client does not contact a server more than once during a transaction. Non-blocking [87] states that servers should not communicate with each other before responding to the client. Finally, one-version asks that a server only sends one value for each read object. In the same work, Lu et al. prove that fast read-only transactions are possible by presenting COPS-SNOW, a causally-consistent data store that provides fast read-only transactions. Because of the critical importance of fast read-only transactions for industrial data stores, fast read-only transactions have received much attention of late [44, 45, 71, 87, 118]. However, all this research [44, 45, 71, 87, 118] targets the ideal case where servers never fail.

In practice, distributed systems are deployed in settings where failures are the norm. Distributed systems ought to be fault-tolerant. Fault-tolerance is usually achieved by means of replication or logging. However, replication and logging are synchronous (i.e., blocking) operations. As expected, achieving fault-tolerance has a cost on the performance of transactions that write to objects. Write transactions cannot be fast since they are blocking: write transactions have to replicate data before committing. It is therefore natural that the description of fast transactions [87] only refers to read-only transactions.

In this chapter we prove that, surprisingly, read-only transactions cannot be fast either. Fast transactions in general are impossible in a system that aims to be fault-tolerant. To show that read-only transactions cannot be fast, we examine the concept of the visibility of transactions, and investigate whether fast read-only transactions can be invisible. Transactions are said to be invisible if they do not modify the state of the servers with which they communicate.

²In this context, we use the term “partition” in the sense of sharding.

Intuitively, visible read-only transactions modify the state of the server they are operating on. Thus, visible read-only transactions are blocking, since the modification on the server has to be performed in a fault-tolerant way (e.g., by replicating the modification to other servers). This means, that in a fault-tolerant system, read-only transactions can only be fast if they are invisible.

We prove that in an asynchronous system, if we require transactions to ensure monotonic writes, a minimal level of consistency that is weaker than causal consistency, then read-only transactions cannot be both fast and invisible. In fact, our proof holds for a weaker definition of fast transactions, one where a server can send a bounded number of versions (instead of just one) back to the client. In this sense, we prove a more general result. Our result sheds some light on a so far unexplored limitation of fast transactions: they are impossible if we want to tolerate the failure of even one server. Furthermore, we prove that if a server can send an unbounded number of versions back to the client, then transactions can be non-blocking and one round-trip. To prove this, we devise a new data store algorithm that we believe is interesting in its own right, called `ubvStore`, that provides non-blocking and one round-trip transactions.

The practical implications of our theoretical results are threefold. First, similar to the CAP theorem [27, 56], we demonstrate a new trade-off for data stores: either fast transactions or fault-tolerance can be achieved, even with the weak consistency model of monotonic writes. Second, our results allow designers to avoid chasing impossible designs. Third, we show that fault-tolerance should be a first-class concern when proposing new theoretical properties, in order for the properties to have practical utility.

To prove our results, we devised a new formal framework that is general enough to capture any data store, while at the same time the framework is able to precisely capture notions such as bounded-version, non-blocking, etc., a challenging endeavor. As far as we know, our formalism is the first that precisely captures the notion of fast read-only transactions. We consider the framework as a contribution on its own.

Roadmap. The rest of this chapter is organized as follows. We present our framework in Section 4.2. In Section 4.3, we prove the impossibility of fast transactions and we discuss the ramifications of our result. Then, in Section 4.4, we describe the `ubvStore` algorithm. Finally, in Section 4.5, we discuss related work before concluding.

4.2 Model

We consider an asynchronous model that captures the notion of a data store that supports write operations on single objects, as well as read-only transactions that operate on groups of objects.

Our model does not support transactions that write to objects, hence our impossibility result is stronger. Since we only consider read-only transactions, whenever we refer to a *transaction*, we refer to a read-only transaction. We distinguish between servers and clients in our system and clearly define the ways they can communicate and the exact type of messages a client can send to the server. Nevertheless, the model provides great flexibility on how the processes (i.e., clients and servers) can communicate.

We consider a *data store* as a message-passing system with servers and clients that communicate. Servers store objects that the clients can read or write. Specifically, a data store is a tuple $(\mathcal{S}, \mathcal{C}, \mathcal{O}, \mathcal{V}, \mathcal{M}, dec)$ where $\mathcal{S}, \mathcal{C}, \mathcal{O}, \mathcal{V}$, and \mathcal{M} are sets and dec is a function, as described below. We consider that a data store consists of n servers contained in a set $\mathcal{S} = \{s_1, \dots, s_n\}$, as well as a finite set $\mathcal{C} = \{c_1, c_2, \dots, c_m\}$ of m clients. We consider that both servers and clients are deterministic. Clients with servers, as well as servers with servers communicate by exchanging messages, where messages can take arbitrary time to be delivered but eventually are delivered (i.e., no message is lost). We assume that clients cannot communicate with each other. Additionally, we consider a set $\mathcal{O} = \{o_1, \dots, o_l\}$ of l objects and an infinite set $\mathcal{V} = \{v_1, v_2, \dots\} \cup \{\perp\}$ of finite values that the objects can take. Value \perp corresponds to the initial value of each object. No write operation can write \perp value to an object. Note that depending on the context, we refer interchangeably to a value as a *version*. We also consider an infinite set of messages \mathcal{M} , where each message $m \in \mathcal{M}$ is created over some alphabet. All the infinite sets we consider are countable. Finally, we consider the decoding function $dec: \mathcal{M} \rightarrow 2^{\mathcal{V}}$, that given a message m returns a set of values that are encoded in m . We use function dec to bound the number of values a client can utilize from a given message.

We introduce some notation that we use throughout this chapter. For a set S we define $S^{\leq k} = S^1 \cup S^2 \cup S^3 \cup \dots \cup S^k$, where $S^1 = S$ and for $i > 1$ $S^i = S \times S^{i-1}$. For a tuple $v = (v_1, v_2, \dots, v_g)$ we denote with v_i the i -th element of v , e.g., $(3, 8, 1)_2$ is 8. Finally, given a sequence of elements $\alpha = a_1, a_2, \dots$, we denote with $a_i \in \alpha$ that a_i appears in α .

Server. We model a *server* as a state machine $s = (\Sigma, \sigma_0, E, \Delta, obj)$ where Σ is the set of possible states, $\sigma_0 \in \Sigma$ is the initial state and E is the set of possible events. $\Delta: \Sigma \times E \rightarrow \Sigma$ is a partial function that captures the possible state transitions a server can take based on a given event. The set $obj \subseteq \mathcal{O}$ is the set of objects that the server handles. For a server s , we denote with $s.\Sigma$, $s.\sigma_0$, $s.E$, $s.\Delta$, and $s.obj$ the set of states Σ of server s , the set of events E of server s , etc. We say that a server s *serves* objects $s.obj$ or a server *serves* an object o where $o \in s.obj$. We consider a system where all the objects are served by some server, hence $\bigcup_{i=1}^n s_i.obj = \mathcal{O}$, and additionally we assume that every object is being served by a single server, hence $\forall s, s' \in \mathcal{S}$ with $s \neq s'$ $s.obj \cap s'.obj = \emptyset$.

In what follows, we refer to either a server or a client as a *process*. For each server $s \in \mathcal{S}$, the

set of events is defined as $s.E = \{send(m, p), receive(m, p) : m \in \mathcal{M}, p \in \mathcal{S} \cup \mathcal{C}\}$. Specifically, a $send(m, p)$ event corresponds to server s transmitting message m to process p . Event $receive(m, p)$ corresponds to server s receiving message m from process p .

Client. We model a *client* as a state machine $c = (\Sigma, \sigma_0, E, \Delta, \rho)$ where Σ , σ_0 , and Δ are defined in the same way as of a server. Function ρ captures the notion of the exact values a client reads for a transaction. We consider an infinite set $\mathcal{T} = \{t_1, t_2, \dots\}$ of finite transactional identifiers. Then, function $\rho : \Sigma \times \mathcal{T} \rightarrow \mathcal{V}^{\leq l}$ (note that $|\mathcal{O}| = l$) is given a state $\sigma \in \Sigma$ and a transactional identifier and returns an arbitrary number of values. Function ρ is used to define the monotonic writes consistency property (see later on). Additionally, the set of events E for a client is different than that of a server, since for a client c we restrict the events c can take (i.e., the messages a client can send and receive).

A client can either perform a write operation on a single object, or a transaction on a set of objects that are served by more than one server. To clearly capture the notion of a transaction in our model, we consider that a client splits a transaction into multiple read operations where each read operation is destined to a different server with the objects to be read. Specifically, a client can only issue two kinds of operations, a *read* and a *write* operation. In what follows, we first describe the exact operations a client can issue. We then describe what kind of messages a client c can send and receive (i.e., events a client can take) based on the operations c performs. A $read(O_s, t, d)$ operation reads ℓ objects defined in ℓ -tuple $O_s \in \mathcal{O}^\ell$, and $d \in \mathcal{M}$ as part of some transaction with identifier $t \in \mathcal{T}$. Note that a read operation reads from distinct objects, thus for a read $read(O_s, t, d)$ where $O_s = (o_1, o_2, \dots, o_\ell)$ is an ℓ -tuple, $\forall i, j \in \{1, \dots, \ell\}$ with $i \neq j$, it is the case that $o_i \neq o_j$. A $read(O_s, t, d)$ operation is always part of a *transaction*. Naturally, a client can perform a transaction on objects that reside on different servers. In such a case, a client sends two read operations with the same transactional identifier to two different servers. For example, assume we have two servers $s_1, s_2 \in \mathcal{S}$ with $s_1.obj = \{o_1\}$, $s_2.obj = \{o_2\}$, and a client $c \in \mathcal{C}$ wants to perform a transaction that reads both objects o_1 and o_2 . Then, client c has to issue a $read((o_1), t_{id}, d_1)$ operation to server s_1 and a $read((o_2), t_{id}, d_2)$ to server s_2 .

A $write(o, v, d)$ operation writes value v to single-object o where $o \in \mathcal{O}$, $v \in \mathcal{V} \setminus \{\perp\}$, and $d \in \mathcal{M}$. Note that both the read and write operations take as a parameter a message d . This message is not necessarily bounded (since a message can be of any size) and can contain additional information the client might want to include in its operation. We specifically allow clients to send any message d in order to have a model as general as possible. This way, we do not restrict the possible data stores the model expresses.

Client responses. The response to a $read(O_r, t, d)$ operation with $|O_r| = r$ is $res(x)$ where $x \in \mathcal{O}^r \times \mathcal{M}$. Specifically, $res(x) = (O_r, m)$ where $m \in \mathcal{M}$. In other words, the response to a *read* reading r objects is a pair of one tuple that contains the to-be-read r objects and the message

m containing the values for the r objects. Naturally, a response to a single-object $read(o, t, d)$ is $res(x)$ where $x = (o, m)$ and $m \in \mathcal{M}$. Note that a message m can contain multiple values (i.e., versions) for a specific object. We present later how to use dec to restrict the possible values a client can retrieve from a message.

Similarly to a read operation, we consider that a response to a $write$ operation is $res(d)$ where $d \in \mathcal{M}$. Note that we can get a response $res(d)$ with $d \in \mathcal{M}$ only in response to a write operation.

A client can issue a transaction that consists of multiple read operations. The responses from the read operations are used to extract the values the transaction reads using ρ .

Client events. We describe the set of all messages the client can send or receive. The set of messages the client can send is $\mathcal{M}_s = \{m : m = read(\mathcal{O}_r, t_{id}, d) \text{ and } \mathcal{O}_r \in \mathcal{O}^{\leq l}, t_{id} \in \mathcal{T}, d \in \mathcal{M}\} \cup \{m : m = write(o, v, d) \text{ and } o \in \mathcal{O}, v \in \mathcal{V} \setminus \{\perp\}, d \in \mathcal{M}\}$. The set of messages the client can receive is $\mathcal{M}_r = \{m : m = res(x) \text{ and } x \in \mathcal{O}^{\leq l} \times \mathcal{M}\} \cup \{m : m = res(d) \text{ and } d \in \mathcal{M}\}$. The set of possible events a client $c \in \mathcal{C}$ can take is $c.E = \{send(m_s, p) : m_s \in \mathcal{M}_s, p \in \mathcal{S}\} \cup \{receive(m_r, p) : m_r \in \mathcal{M}_r, p \in \mathcal{S}\}$. Finally, note that clients cannot communicate with each other but only with servers, a natural assumption [9, 86, 87]. It might seem that a client can issue a read or a write operation for an object o to a server s that does not serve o . We restrict these cases later, when we define what a well-formed execution is.

Execution. We say that an event e is *enabled* in state σ if $\Delta(\sigma, e)$ is defined. An *execution* is a (possibly infinite) sequence of events occurring at the servers and the clients. A sequence of events occurring at a process p (i.e., p is either a server or a client) is *well-formed* if there is a sequence of states, $\sigma_1, \sigma_2, \dots$ such that $\sigma_i = p.\Delta(\sigma_{i-1}, e_i)$ for all $2 \leq i \leq$ (the length of the sequence). An execution has *correct issues* of operations if every $read(\mathcal{O}_r, t, d)$ with $\mathcal{O}_r = (o_1, \dots, o_\ell)$ that a client issues is destined to a server s where $\forall i, 1 \leq i \leq \ell, o_i \in s.obj$, as well as every $write(o, v, d)$ operation is destined to a server s where $o \in s.obj$. We assume that clients have some initial knowledge on which server contains which objects, a reasonable assumption in practice since such information could be stored in the initial state of each client.

We say that an event e is a *client write request* if e corresponds to the *send* event of a client for a write operation. We say that an event e is a *client read request* if e corresponds to the *send* event of a client for a read operation. For example, event $send(read(\mathcal{O}_r, t, d), s)$ taken by some client is a client read request, while event $send(write(o, v, d), s)$ is a client write request. We say that an event e is a *client request* if e is a client read or write request. Similarly, we say that an event e is a *client read response* if e corresponds to the receipt event of a client with $res(x)$ and $x \notin \mathcal{M}$ (i.e., $e = receive(res(\mathcal{O}_r, m), c)$). We call an event e a *client write response* if e corresponds to the receipt event of client with a $res(d)$ event where $d \in \mathcal{M}$. We say that an event e is a *client response* if e is a client read or write response. For brevity, we also use

the notation $e = \text{read}(O_s, t, d)$, $e = \text{write}(o, v, d)$, or $e = \text{res}(x)$, when it is clear from the context whether event e is being sent or received by a client or by a server.

For a given client request e we define $\text{obj}(e)$ to be the tuple of objects e is operating on. For example, if e is a $\text{read}((o_5, o_8), t, d)$, then $\text{obj}(e) = (o_5, o_8)$. Similarly, for a client read request e that is associated with a transaction, $\text{tx}(e)$ provides the transactional identifier associated with e . Again, if an event e is taken by a client, we denote with $\text{cl}(e) \in \mathcal{C}$ the client that took e . For every process $p \in \mathcal{C} \cup \mathcal{S}$, given an execution α , we define the *process execution* $\alpha|_p$ to be the subsequence of α that contains all the events of α taken by process p . Given an execution α , we define the *read execution* $\alpha|_{\text{read}}$ to be the subsequence of α containing only client read requests and client read responses. Similarly, we define execution $\alpha|_{\text{write}}$ to be the subsequence of α containing only client write requests and client write responses.

Valid responses. An execution α has *no-thin-air responses*, if for every client $c \in \mathcal{C}$, for every client response event $e' = \text{res}(x)$ in $\alpha|_c$, there is a client request event e that precedes e' in $\alpha|_c$ such that e is either a *write* event if $x \in \mathcal{M}$, or e is a $\text{read}(O_s, t, d)$ event if $x = (O_s, m)$ with $O_s \in \mathcal{O}^{\leq l}$ and $m \in \mathcal{M}$. As the name suggests, no-thin-air responses captures the notion that client responses are not created out of thin-air (i.e., a client request should trigger the response).

An execution α has *written-values responses*, if for every client $c \in \mathcal{C}$, for every client read response event $e' = \text{res}(x)$ with $x = (O_s, m)$, then for every $v \in \text{dec}(m)$, there should be an object $o \in O_s$ such that there is a client write request $e = \text{write}(o, v, d)$ that appears before e' in α . In other words, the values a client is reading were written by some server at some previous point in time.

An execution α has *valid responses* if α has no-thin-air and written-values responses.

Sequential clients. Clients can issue reads as part of the same transaction to objects belonging to different servers. For example, a client might issue a transactional $\text{read}((o_1, o_2), t, d)$ to a server s_1 and another read $\text{read}((o_4, o_5), t, d)$ to a server s_2 . Clients are said to be *sequential*. This means that a client can issue a write or a read operation only if the client has received responses to all its previous requests. Furthermore, a client c can issue a read with a transactional identifier t if c has received responses to a previous write request, as well as to all transactional reads of a transaction t' with $t' \neq t$. In other words, a client c can issue parallel read operations for the same transaction, but c has to wait for a response to its previous operations before issuing a write or a new transaction.

Formally, we say that an execution α has *sequential clients* if for every client $c \in \mathcal{C}$, for every client request $e \in \alpha|_c$, where e is not the last event in $e \in \alpha|_c$, the following holds:

- if $e = \text{read}(O_r, t, d)$, then the event e' that immediately succeeds e in $\alpha|_c$ has $tx(e') = tx(e)$ or $e' = \text{res}(x)$ with $x = (O_{r'}, m)$ with $m \in \mathcal{M}$ ($O_{r'}$ is not necessarily equal to O_r);
- if $e = \text{write}(o, v, d)$, then the event e' that immediately succeeds e is $\text{res}(d)$ with $d \in \mathcal{M}$.

Valid values. Next, we provide auxiliary definitions that help us capture the notion of a bounded-version data store.

Definition 3 (Corresponding event). *Given an execution α that has no-thin-air responses, consider a client response $e' = \text{res}(x)$ where $e' \in \alpha$. Event e' is received by client $c \in \mathcal{C}$ in response to client's c request e . We say that event e' has e as its corresponding event, or that response e' has e as its corresponding request. Conversely, request e has e' as its corresponding client response.*

Given a client request e and its corresponding client response e' in an execution α , we denote e' 's corresponding client response with $\text{cor}(e)$. We say that a client request e is *completed* in an execution α if $\text{cor}(e) \in \alpha$. Note that a transaction can be split into many client read requests and the completion of some of these requests does not imply that the transaction has completed.

We say that a client *read response* e is *associated with a transaction* t if e 's corresponding read request e' has $tx(e') = t$. A transaction t is *completed* in an execution α if there is a read request event $e \in \alpha$ with $tx(e) = t$ and $cl(e) = c$ and there is a read request event e' that succeeds e in $\alpha|_c$ such that $tx(e') \neq t$. Given an execution α , we define as $\text{comp}(\alpha) \subseteq \mathcal{T}$ the set of all completed transactions in α .

Definition 4 (Last state of a transaction). *Consider an execution α and a transaction $t \in \text{comp}(\alpha)$ issued by a client c , we denote with $\sigma_{\text{last}}(t, \alpha)$ the last state of client c that was part of transaction t . $\sigma_{\text{last}}(t, \alpha)$ corresponds to the state immediately after the last event associated with t took place in α by c .*

The following definition helps us define correctness in an execution on what a transaction reads. For this, we need to know what values are read by a transaction.

Definition 5 (Values of a transaction). *Given an execution α , we say a transaction $t \in \text{comp}(\alpha)$ reads values $\rho(\sigma_{\text{last}}(t, \alpha), t)$.*

Consider an execution α and consider a client $c \in \mathcal{C}$, we define as $\text{msg}(\alpha, c)$ the set of messages contained in all the client responses in $(\alpha|_{\text{read}})|_c$.

The definition below captures the notion that a transaction by a client can only read the initial value (\perp) or values that were at some point received by a server.

Definition 6 (Valid values). *Given an execution α and a transaction $t \in \text{comp}(\alpha)$, we say that t reads valid values if for every $v \in \rho(\sigma_{\text{last}}(t, \alpha), t)$, there is an $m \in \text{msg}(t, \alpha)$ such that $v \in \text{dec}(m)$.*

Well-formed execution. An execution α has *distinct values* if for every two write operations, $\text{write}(o, v, d)$ and $\text{write}(o', v', d')$ where $o, o' \in \mathcal{O}$, $v, v' \in \mathcal{V}$, $d, d' \in \mathcal{M}$, it is the case that $v \neq v'$. We can achieve this in practice by having a client append its client identifier and a monotonically increasing counter to the value it intends to write. We say that an execution has *no-transaction reuse* when each client c uses different transactional identifiers for each of c 's transactions, as well as different transactional identifiers from other clients.

An execution α is *well-formed* if the following conditions hold for α :

- $\forall p \in \mathcal{S} \cup \mathcal{C}$, $\alpha|_p$ is well-formed;
- α has correct issues, valid responses, sequential clients, distinct values, and no-transaction reuse;
- for every $t \in \text{comp}(\alpha)$, transaction t reads valid values;
- if there is a $\text{receive}(m, p_j)$ event e' taken by some process p_i in α , then there is an event e that precedes e' in α and $e = \text{send}(m, p_j)$ by process p_j ;
- for a specific $m \in \mathcal{M}$ if there are z identical events $\text{send}(m, p_j)$ taken by process p_j in α , then there are at most z $\text{receive}(m, p_j)$ events taken by process p_i in α .

The last two conditions state that a message is not received out of thin air and that there is no message duplication. Both conditions can be implemented in practice with common techniques, such as the use of timestamps [33]. In this chapter and unless stated otherwise, we consider only well-formed executions. When we talk about an implementation in our model, we refer to the state machines of all the servers (i.e., function Δ) and all the clients, as well as the sets \mathcal{S} , \mathcal{C} , \mathcal{O} , \mathcal{V} , \mathcal{M} , and function dec . We denote an *implementation* with \mathcal{I} and say that $\alpha \in \mathcal{I}$ to denote that execution α can be generated by implementation \mathcal{I} . Note that if a data store \mathcal{I} can generate an execution α' , \mathcal{I} can also generate any execution α where α is a contiguous prefix of α' . Formally, a data store is:

Definition 7 (Data store). *A data store is an implementation \mathcal{I} such that for every execution $\alpha \in \mathcal{I}$, α is a well-formed execution.*

Bounded-version data store. In response to a client's read operation to an object o , a server can potentially send an unbounded number of values that were written to o back to the client.

Chapter 4. The Impossibility of Fast Transactions

In this work, we prove that non-blocking and one round-trip transactions are impossible when a server can only send a bounded number of values to a client. Therefore, we need to define what it means for a data store to be bounded-version. Instead of restricting the number of values a server can send to a client, we allow a server to send an unbounded number of values, and then use (among others) the *dec* function to restrict the number of values a client can utilize.

Definition 8 (*k*-version data store). *We say that a data store \mathcal{J} is *k*-version if for every $m \in \mathcal{M}$, $|\mathcal{J}.dec(m)| \leq k$.*

Although messages are finite, they are unbounded, hence the above definition allows a server to send back an arbitrary long message $m \in \mathcal{M}$ to the client, and hence an unbounded number of values to a client. This allows a client to cache old values and use them in future transactions. However, in combination with valid values (Definition 6) the client can only extract up to a bounded number of values. Note that even if a client uses a cache to store retrieved values, these values cannot be used by the client unless they belong to a decoding of an already received message.

If for a data store \mathcal{J} , there is a $k > 0 \in \mathbb{N}$ such that \mathcal{J} is a *k*-version data store, then we say that \mathcal{J} is a *bounded-version data store*, otherwise we say that \mathcal{J} is an *unbounded-version data store*. To the best of our knowledge, function *dec* is the first one that is able to formally capture the notion of a *k*-version data store in an elegant way. Other approaches [44, 45, 71] are not formal enough and can be potentially circumvented (see Section 4.5).

Fast reads. In what follows, we define the notion of invisible reads, which refers to the fact that servers do not update their state when they perform a read operation.

Definition 9 (Invisible reads). *A data store \mathcal{J} has invisible reads, if for every execution $\alpha \in \mathcal{J}$, for every received read request event or every sent read response event e by some server $s \in \mathcal{S}$, $\sigma = s.\Delta(\sigma, e)$.*

Definition 9 captures the fact that if the state of the server s is σ before event e , then it remains σ after event e takes place. A data store \mathcal{J} that does not provide invisible reads, is said to have *visible reads*. Next, we define non-blocking reads.

We consider function *nc* that given an execution α and an event $e \in \alpha$ returns a subsequence of α . Formally, consider an execution α , a client c , and client response $e' \in \alpha|_c$, then $nc(\alpha, e')$ corresponds to execution α where all the events between the corresponding request e of e' to a server s and e' are removed from α , except the events that correspond to server s receiving request e and responding e' back to c .

Intuitively, a data store supports non-blocking reads if a server can respond to a read operation without blocking. This means that the server does not have to communicate with other servers in order to respond to the client. Formally:

Definition 10 (Non-blocking reads). *We say that a data store \mathcal{S} has non-blocking reads, if for every finite execution $\alpha \in \mathcal{S}$ that ends in a client read response e , then $nc(\alpha, e) \in \mathcal{S}$.*

We now define what it means for a transactional read to take a specific number of rounds. Roughly speaking, an operation takes r rounds, if a clients performs r client responses (i.e., received from the same server) for one client read request.

Definition 11 (r -round reads). *We say that a data store \mathcal{S} has r -round reads, if for every execution $\alpha \in \mathcal{S}$, every transaction $t \in \mathcal{T}$, every client c performs at most r client read responses for a specific read request associated with transaction t .*

For instance, in a data store that has 1-round reads this means that a client that issues a transaction t only communicates with a specific server at most once for transaction t .

Definition 12 (Fast reads). *We say that a data store \mathcal{S} has fast reads if \mathcal{S} is a 1-version data store, and \mathcal{S} has non-blocking and 1-round reads.*

Definition 12 captures the notion of latency-optimal reads as informally described by Lu et al. [87], since a client can utilize only one value per read object. We relax the definition of fast reads, by defining what we call semi-fast reads.

Definition 13 (Semi-fast reads). *We say that a data store \mathcal{S} has semi-fast reads if \mathcal{S} is a bounded-version data store, and \mathcal{S} has non-blocking and 1-round reads.*

In contrast to fast reads, semi-fast reads allow a server to send more than one value back to a client in response to a read request. In this sense, semi-fast reads are not latency-optimal as devised by Lu et al. [87]. We prove our impossibility result for semi-fast reads and hence our impossibility result is stronger (i.e., also holds for fast reads).

Monotonic writes. We consider data stores that provide the client-centric consistency model [119] of monotonic writes [116].

Given an execution α and a client $c \in \mathcal{C}$, we define with $ord_w(\alpha, c)$ the set of pairs (e, e') such that e and e' are in $(\alpha|_c)|_{write}$ and e precedes e' in α (and hence in $\alpha|_c$). We define with $ord_w^+(\alpha, c)$ the transitive closure of $ord_w(\alpha, c)$.

Roughly speaking, a data store provides monotonic writes consistency if the write operations performed by a specific client in some specific order, are seen by any other client in this order.

Note that monotonic writes do not specify anything regarding the order of write operations between different clients. We use the notation $t \in \alpha$ to denote that there is an event $e \in \alpha$ such that $tx(e) = t$.

Definition 14 (Monotonic writes.). *Consider an execution α and consider every transaction $t \in \text{comp}(\alpha)$ with values $\rho(\sigma_{\text{last}}(t, \alpha), t) = (v_1, v_2, \dots, v_r)$. Values (v_1, v_2, \dots, v_r) are written by client write requests that correspond to events $e_{w_1}, e_{w_2}, \dots, e_{w_r}$. We say that α provides monotonic writes if for any two client write requests e_{w_i} and e_{w_j} with $c = cl(e_{w_i}) = cl(e_{w_j})$ there is no client write request e_w with $cl(e_w) = c$ such that $(e_{w_i}, e_w) \in \text{ord}_w^+(\alpha, c)$ and $(e_w, e_{w_j}) \in \text{ord}_w^+(\alpha, c)$ and $\text{obj}(e_{w_i}) = \text{obj}(e_w)$.*

Figure 4.1 depicts the intuition behind Definition 14. It should not be possible for a transaction reading, among others, objects o and o' to read values v_i and v_j , since the same client wrote value v to object o after writing value v_i to o .

$$\text{client } c: \quad e_{w_i}(o = v_i) \longrightarrow e_w(\cancel{o = v}) \longrightarrow e_{w_j}(o' = v_j)$$

Figure 4.1 – A transaction t that reads, among others, objects o and o' cannot read values v_i and v_j due to the existence of e_w . However, transaction t can read values v and v_j for objects o and o' respectively. Note that all three writes e_{w_i} , e_w , and e_{w_j} are performed by the same client c .

Definition 15 (Monotonic writes). *We say that a data store \mathcal{J} provides monotonic writes if every execution $\alpha \in \mathcal{J}$ has monotonic writes.*

Note that current definitions of monotonic writes [29, 116] refer to single-object operations (i.e., no transactions). Bailis et al. [17] provide an intuitive transactional description but lacks adequate formality. In this sense, this is the first formal definition of monotonic writes in a transactional setting.

Minimal progress. A data store implementation where every read operation performed by a client reads back the initial value of the object \perp is a data store that provides monotonic writes. However, such a data store is of no practical interest. We need to incorporate some notion of liveness in the data store to make it useful. For this, we introduce the notion of minimal progress, that roughly speaking states that if only one client writes a value v to an object o , this value is visible after some some point in time, unless the same client writes a new value or some other client writes o . We start by defining the notion of being eventually responsive. The intuition behind this definition, is that if a client requests a read or write operation, then the client eventually gets a response.

Definition 16 (Eventually Responsive). *A data store \mathcal{J} is eventually responsive if for every*

finite execution $\alpha \in \mathcal{J}$ with last event e that is a client request, every infinite extension $\alpha' \in \mathcal{J}$ of α has a client response e' such that e' 's corresponding request is e .

Another way to think of the above definition is that given a finite execution α that ends with a request to a client, there is an extension of α that contains the response to the client.

We say that a *transaction t appears for the first time* after an event e' in an execution α if there is no event $e \in \alpha$ that appears before e' in α with $tx(e) = t$. The following definition captures the idea that from some point onwards, all transactions keep returning the newer written values. If a single write with value v is performed to an object o , then there is a point after which all transactions that appear for the first time read v or a later written value for object o .

Definition 17 (Bounded Visible). *A data store \mathcal{J} is called eventually visible if for every finite execution $\alpha \in \mathcal{J}$ with last event e_w that is a client write request to object o . Consider all $r > 0$ completed client write requests before e_w to o : e_{w_1}, \dots, e_{w_r} in α . In other words, $cor(e_{w_1}), \dots, cor(e_{w_r})$ appear before e_w in α . Each of these client write requests, writes a value. Consider these values to be contained in the set v_{old} . Then, there is a bound $b \geq 0$, such that for every extension $\alpha' \in \mathcal{J}$ of α , every transaction that appears for the first time after $|\alpha| + b$ in α' and that requests o , does not read a value that belongs to $v_{old} \cup \{\perp\}$ for o .*

Definition 18 (Minimal Progress). *A data store \mathcal{J} provides minimal progress if \mathcal{J} is eventually responsive and bounded visible.*

Note that if a data store \mathcal{J} provides minimal progress, \mathcal{J} cannot use the stable snapshot approach [118]. In the stable snapshot approach, servers totally order write operations, as well as servers keep track of the most recent stable snapshot. A stable snapshot is a point in the serial order of the write operations, for which all updates are known to have been applied to all objects. When reading, a client c indicates the last known stable snapshot, and then c reads from that snapshot and retrieves information about the current last known stable snapshot (that c uses in the next transaction). Thus client c can always make progress, albeit by reading from the past. However, using the stable snapshot approach, when a client c that has been inactive (i.e., not performing any operations) for an arbitrary long amount of time, issues a new transaction, c could potentially read old values and violate bounded visibility.

4.3 Fast Transactions Are Impossible

In this section, we prove that fast transactions are impossible. Specifically, we prove our main result.

Theorem 5. *No data store can provide monotonic writes, minimal progress, and invisible semi-fast reads.*

Fast reads. For pedagogical purposes, we first present the proof of the following theorem:

Theorem 6. *No data store can provide monotonic writes, minimal progress, and invisible fast reads.*

For our result, we consider two servers $s_1, s_2 \in \mathcal{S}$ and two objects $o_1, o_2 \in \mathcal{O}$ served by servers s_1 and s_2 respectively. Furthermore, we consider three clients $c_r, c_h, c_w \in \mathcal{C}$, where client c_h issues a finite number of transactions where each transaction reads both objects o_1 and o_2 , client c_r issues a single transaction t to both objects o_1 and o_2 , and client c_w performs writes. Before we continue, we introduce some auxiliary notation to simplify the proof.

$$\begin{aligned} \alpha_1: & w_{o_1}(v_{o_1}^1) \rightarrow w_{o_2}(v_{o_2}^1) \rightarrow [t_1 \rightarrow \dots \rightarrow t_{l_1}] \rightarrow w_{o_1}(v_{o_1}^2) \rightarrow w_{o_2}(v_{o_2}^2) \rightarrow [t_{l_1+1} \rightarrow \dots \rightarrow t_{l_2}] \rightarrow req_{o_1} \rightarrow req_{o_2} \rightarrow resp_{o_1}(m_1^1) \rightarrow resp_{o_2}(m_2^1) \\ \alpha_2: & w_{o_1}(v_{o_1}^1) \rightarrow w_{o_2}(v_{o_2}^1) \rightarrow [t_1 \rightarrow \dots \rightarrow t_{l_1}] \rightarrow req_{o_1} \rightarrow req_{o_2} \rightarrow resp_{o_1}(m_1^2) \rightarrow w_{o_1}(v_{o_1}^2) \rightarrow w_{o_2}(v_{o_2}^2) \rightarrow [t_{l_1+1} \rightarrow \dots \rightarrow t_{l_2}] \rightarrow resp_{o_2}(m_2^2) \end{aligned}$$

Figure 4.2 – Executions α_1 and α_2 where client c_w writes (in red) objects o_1 and o_2 . Client c_h issues a finite number of transactions (in green) between the c_w 's writes and c_r performs a transaction that reads both objects o_1 and o_2 (in blue).

A read operation in a data store with fast reads consists of 4 events e_1, e_2, e_3, e_4 in this order. Event e_1 is $send(read(o, t_i, m_s), s)$, event e_2 is $receive(read(o, t_i, m_s), c)$, and since a data store with fast reads is non-blocking, this means the server s can respond right away to the client with $e_3 = send(res((o, m_r)), c)$, and finally the client c receives the message $e_4 = receive(res((o, m_r)), s)$. In what follows we are going to denote event e_1 as req_o and the remaining three events as $resp_o(m_r)$, meaning that for object o the client got back response m_r .

Similarly, we denote with $w_o(v)$ the sequence of events needed to write value v to object o . In contrast to a read operation event, a write operation event might span an arbitrary, however finite, number of events. Arbitrary since write operations are not necessarily non-blocking and hence a server might communicate with other servers before completing the write. Finite since we consider a data store with eventual responsiveness (i.e., the write should eventually complete).

For the proof of Theorem 6 we assume by way of contradiction that a data store exists that provides monotonic writes, minimal progress, and invisible fast reads. For this, we consider execution:

$$\alpha = w_{o_1}(v_{o_1}^1), w_{o_2}(v_{o_2}^1), t_1, \dots, t_{l_1}, w_{o_1}(v_{o_1}^2), w_{o_2}(v_{o_2}^2), t_{l_1+1}, \dots, t_{l_2}$$

where the first two writes are performed by client c_w and transactions t_1, \dots, t_{l_1} are performed by client c_h until values $v_{o_1}^1$ and $v_{o_2}^1$ become visible. Then, client c_w performs two more writes to objects o_1 and o_2 and afterwards client c_h takes steps until values $v_{o_1}^2$ and $v_{o_2}^2$ are visible.

4.3. Fast Transactions Are Impossible

Based on execution α , we construct executions α_1 and α_2 :

$$\alpha_1 = w_{o_1}(v_{o_1}^1), w_{o_2}(v_{o_2}^1), t_1, \dots, t_{l_1}, w_{o_1}(v_{o_1}^2), w_{o_2}(v_{o_2}^2), t_{l_1+1}, \dots, t_{l_2}, req_{o_1}, req_{o_2}, resp_{o_1}(m_1^1), resp_{o_2}(m_2^1)$$

$$\alpha_2 = w_{o_1}(v_{o_1}^1), w_{o_2}(v_{o_2}^1), t_1, \dots, t_{l_1}, req_{o_1}, req_{o_2}, resp_{o_1}(m_1^2), w_{o_1}(v_{o_1}^2), w_{o_2}(v_{o_2}^2), t_{l_1+1}, \dots, t_{l_2}, resp_{o_2}(m_2^2)$$

Note the differences between executions α_1 and α_2 (see Figure 4.2). In execution α_1 , client c_r performs both read operations on objects o_1 and o_2 when the values $v_{o_1}^2$ and $v_{o_2}^2$ are visible. In execution α_2 , c_r read requests are sent after values $v_{o_1}^1$ and $v_{o_2}^1$ are visible. However, the request to object o_1 is received by s_1 before value $v_{o_2}^2$ is visible and the request to object o_2 is received by s_2 after value $v_{o_2}^2$ is visible.

In execution α_1 client c_r receives only two messages, m_1^1 and m_2^1 . Message m_1^1 cannot contain a value for object o_2 since o_2 is not served by server s_1 and s_1 can only contain values for object o_1 . Therefore, for c_r to read value $v_{o_2}^2$ for object o_2 , it should be the case that message m_2^1 in execution α_1 contains $v_{o_2}^2$. Assume by way of contradiction that m_2^1 does not contain $v_{o_2}^2$, this means that there is no way for client c_r to have read value $v_{o_2}^2$, therefore $v_{o_2}^2 \notin \rho(\sigma_{last}(t, \alpha_1), t)$, hence the value is not visible yet. A contradiction, therefore $v_{o_2}^2 \in dec(m_2^1)$.

In execution α_2 note that m_1^2 cannot contain $v_{o_1}^2$ (due to the written-values responses property) since at this point in execution α_2 , $v_{o_1}^2$ has not been written yet. We argue that message m_2^2 in execution α_2 should contain value $v_{o_2}^1$. Assume it does not. Then, the only other possible values m_2^2 can contain are \perp and $v_{o_2}^2$ (recall that we consider fast reads, hence 1-version reads). However the pair $(v_{o_1}^1, \perp)$ would not satisfy minimal progress since value $v_{o_2}^1$ is visible when client c_r performed its transaction. Furthermore, pair $(v_{o_1}^1, v_{o_2}^2)$ violates monotonic writes, since the same client wrote $v_{o_1}^2$ to object o_1 before writing $v_{o_2}^2$ to object o_2 . A contradiction in both cases. Therefore $v_{o_2}^1 \in dec(m_2^2)$.

Executions α_1 and α_2 are indistinguishable to server s_2 . Indeed, in both executions α_1 and α_2 , server s_2 receives, performs, and responds to the client's c_r request ($resp_{o_2}$) to read object o_2 after values $v_{o_1}^2$ and $v_{o_2}^2$ have been written and are visible. Since all the transactions performed by client c_h are invisible, server s_2 cannot distinguish between the executions and respond back to client c_r in the same way in both executions, and therefore messages m_2^2 , and m_2^1 are equal ($m_2^1 = m_2^2 = m_2$). Since we consider distinct values, $v_{o_2}^1 \neq v_{o_2}^2$, and both are in $dec(m_2)$, it is the case that $|dec(m_2)| = 2 > 1$, a contradiction. Hence, Theorem 6 holds.

Semi-fast reads. We prove Theorem 5 by contradiction. We assume that a data store \mathcal{S} exists that provides monotonic writes, minimal progress, and invisible semi-fast reads. Specifically, we assume that data store \mathcal{S} is a k -version data store. We prove that there is an execution where server s_2 sends a message back to a client that contains more than k values in order for \mathcal{S} to satisfy monotonic writes. To prove our impossibility result, we construct $k + 1$ executions

Chapter 4. The Impossibility of Fast Transactions

$\alpha_1, \alpha_2, \dots, \alpha_{k+1}$ and show that all these executions are indistinguishable to server s_2 . We show that in execution α_i , server s_2 needs to respond with a message that contains value $v_{o_2}^i$.

Before presenting the construction of executions α_i ($1 \leq i \leq k+1$), we first construct execution $\alpha \in \mathcal{J}$ in which all executions α_i are based upon. We construct execution α as follows. First, client c_w writes value $v_{o_1}^1$ to object o_1 , waits for the response of server s_1 to acknowledge the write, and then writes value $v_{o_2}^1$ to object o_2 and waits for server s_2 to acknowledge the write. Servers s_1 and s_2 acknowledge the writes since \mathcal{J} provides minimal progress, and hence \mathcal{J} is eventually responsive. Afterwards, client c_h issues transactions until values $(v_{o_1}^1, v_{o_2}^1)$ are visible, again due to minimal progress. After the values are visible, we allow c_w to write value $v_{o_1}^2$ to object o_1 and afterwards value $v_{o_2}^2$ to object o_2 . Then, we allow client c_h to issue transactions until values $(v_{o_1}^2, v_{o_2}^2)$ are visible. We keep repeating the same procedure until values $(v_{o_1}^{k+2}, v_{o_2}^{k+2})$ are visible. Namely c_w writes both objects o_1 and o_2 and then c_h issues transactions until the latest written values by c_w become visible. Execution α (see top of Figure 4.3) is therefore:

$$\alpha = w_{o_1}(v_{o_1}^1), w_{o_2}(v_{o_2}^1), t_1, \dots, t_{l_1}, w_{o_1}(v_{o_1}^2), w_{o_2}(v_{o_2}^2), t_{l_1+1}, \dots, t_{l_2}, w_{o_1}(v_{o_1}^3), w_{o_2}(v_{o_2}^3), \dots, \\ w_{o_1}(v_{o_1}^{k+2}), w_{o_2}(v_{o_2}^{k+2}), t_{l_{k+1}+1}, \dots, t_{l_{k+2}}$$

$$\alpha: \quad w_{o_1}(v_{o_1}^1) \rightarrow w_{o_2}(v_{o_2}^1) \rightarrow [t_1 \rightarrow \dots \rightarrow t_{l_1}] \rightarrow w_{o_1}(v_{o_1}^2) \rightarrow w_{o_2}(v_{o_2}^2) \rightarrow \dots \rightarrow w_{o_1}(v_{o_1}^{k+2}) \rightarrow w_{o_1}(v_{o_1}^{k+2}) \rightarrow [t_{l_{k+1}+1} \rightarrow \dots \rightarrow t_{l_{k+2}}]$$

$$\alpha_i: \quad w_{o_1}(v_{o_1}^1) \rightarrow w_{o_2}(v_{o_2}^1) \rightarrow \dots \rightarrow w_{o_1}(v_{o_1}^i) \rightarrow w_{o_2}(v_{o_2}^i) \rightarrow [\] \rightarrow req_{o_1} \rightarrow req_{o_2} \rightarrow resp_{o_1}(m_1^i) \rightarrow \dots \\ \dots \rightarrow w_{o_1}(v_{o_1}^{k+2}) \rightarrow w_{o_2}(v_{o_2}^{k+2}) \rightarrow [\] \rightarrow resp_{o_2}(m_2^i)$$

Figure 4.3 – At the top, we depict execution α where client c_w alternates between writing (in red) objects o_1 and o_2 . Client c_h issues a finite number of transactions (in green) after client's c_w write of value $v_{o_2}^j$ until values $(v_{o_1}^j, v_{o_2}^j)$ are visible. At the bottom, we depict execution α_i . Due to space constraints, we depict the transactions of c_h until values $(v_{o_1}^i, v_{o_2}^i)$ and $(v_{o_1}^{k+2}, v_{o_2}^{k+2})$ are visible with shortened green boxes. The events of client's c_r transaction that read objects o_1 and o_2 are depicted in blue.

Before we continue with each individual execution α_i , we prove the following lemma for transactions in execution α , where we consider that $v_{o_1}^0 = v_{o_2}^0 = \perp$.

Lemma 21. *No transaction introduced in execution α that appears for the first time after transaction t_{l_i} ($i > 0$) completes can read values $v_{o_1}^j$ and $v_{o_2}^j$ with $0 \leq j < i$ for objects o_1 and o_2 respectively.*

Proof. By construction of execution α , we know that after transaction t_{l_i} , values $v_{o_1}^i$ and $v_{o_2}^i$ are visible. By the definition of minimal progress, no transaction that appears for the first time

after transaction t_{l_i} has completed can return an older value for objects o_1 and o_2 . Hence no transaction after t_{l_i} can read values $v_{o_1}^j$ and $v_{o_2}^j$ with $0 \leq j < i$. \square

Lemma 22. *No transaction introduced in α that reads objects o_1 and o_2 can read values $(v_{o_1}^a, v_{o_2}^b)$ with $a < b$.*

Proof. Assume by way of contradiction that we can introduce a transaction t in α that reads $(v_{o_1}^a, v_{o_2}^b)$ with $a < b$. If $a < b$, then since client c_w issues writes in this order $v_{o_1}^a \rightarrow \dots \rightarrow v_{o_1}^b \rightarrow v_{o_2}^b$, transaction t that reads $(v_{o_1}^a, v_{o_2}^b)$ with $a < b$ violates monotonic writes, a contradiction, since t should have read value $v_{o_1}^b$ for o_1 .³ \square

We construct execution α_i for $1 \leq i \leq k+1$ as follows (with $l_0 = 0$):

$$\alpha_i = w_{o_1}(v_{o_1}^1), w_{o_2}(v_{o_2}^1), \dots, w_{o_1}(v_{o_1}^i), w_{o_2}(v_{o_2}^i), t_{l_{i-1}+1}, \dots, t_{l_i}, req_{o_1}, req_{o_2}, resp_{o_1}(m_1^i), \\ w_{o_1}(v_{o_1}^{i+1}), w_{o_2}(v_{o_2}^{i+1}), \dots, w_{o_1}(v_{o_1}^{k+2}), w_{o_2}(v_{o_2}^{k+2}), t_{l_{k+1}+1}, \dots, t_{l_{k+2}}, resp_{o_2}(m_2^i)$$

Lemma 23. *Value $v_{o_2}^i \in dec(m_2^i)$ in execution α_i .*

Proof. Assume by way of contradiction that $v_{o_2}^i \notin dec(m_2^i)$ in execution α_i . Due to Lemma 21, transaction t can only read values $(v_{o_1}^a, v_{o_2}^b)$ with $a \geq i$. Since the read to o_1 is performed before the write of value $v_{o_1}^{i+1}$ takes place, this means that t should read value $v_{o_1}^i$ for object o_1 . Therefore the only allowed values for object o_2 is $v_{o_2}^i$ since returning $v_{o_2}^j$ with $j < i$ implies that $v_{o_2}^i$ is not visible yet (Lemma 21) and returning $v_{o_2}^j$ with $j > i$ violates monotonic writes (Lemma 22). In other words, if $v_{o_2}^i \notin dec(m_2^i)$, transaction t has no correct values to return. A contradiction. \square

Due to Lemma 23, we know that $v_{o_2}^i \in dec(m_2^i)$ for every i , $1 \leq i \leq k+1$. However, all executions α_i are indistinguishable to server s_2 and therefore $m_2 = m_2^i$ for every $1 \leq i \leq k+1$. Due to the distinct values property, all values $v_{o_2}^i \neq v_{o_2}^j$ for $i \neq j$ and hence $v_{o_2}^i \in dec(m_2)$ implies that $|dec(m_2)| > k$, a contradiction.

Circumventing the impossibility. In a synchronous system, where the duration of a transaction cannot span more than one synchronous round, Theorem 5 collapses. Meaning that in a synchronous system, we can have a data store with fast invisible reads that also provides minimal progress and monotonic writes. In such a scenario, every server stores the latest value written to an object. In each round, in case of a read, the server returns this latest

³With $v_a \rightarrow v_b$ we denote that a client first performs the write of value v_a , and afterwards the write of value v_b .

value, and in case of a write it overwrites the currently stored value. However, highly available data stores [4, 70, 74] are deployed in the wide-area (i.e., not synchronous setting), since synchronous settings are not realistic, because among others, they prevent the possibility of partitions. Additionally, in a synchronous system we need to choose the duration of rounds in a conservative manner which contradicts the idea of having transactions as fast as possible.

Fast transactions and fault-tolerance. At first sight, the ramifications of Theorem 5 are inconspicuous. Someone might think that whether transactions are visible or not has little impact on the latency of transactions. This might be the case if we assume that the client communicates with a distant server. Nevertheless, in practice, we have to consider the possibility that if a visible transaction updates the state of a server s , s might fail (i.e., crash). In case server s fails, we might lose all the information that was written by s during the update. Highly available data stores have to implement fail-over, therefore, in case server s fails, another server s' should take over in place of s . However, server s' does not contain the data that was written during the update. The only way for server s' to know about the updated state of s , is if s replicated the update to s' before failing. In such a scenario, the transaction is blocking since it has to wait for the underlying writes, to be replicated before responding to the client. To summarize, a data store with fast transactions cannot be fault-tolerant, and conversely, a fault-tolerant data store cannot provide fast transactions.

4.4 Unbounded-Version Data Store

Theorem 5 states that we cannot devise a bounded-version data store with non-blocking and 1-round reads. This raises the question on whether we can achieve non-blocking and 1-round reads in an unbounded-version data store. With Theorem 7 we answer this question in the affirmative.

Theorem 7. *There is an unbounded-version data store that provides monotonic writes, minimal progress, non-blocking and 1-round reads.*

To prove Theorem 7, we devise `ubvStore`, a new unbounded-version data-store with non-blocking and 1-round reads. `ubvStore` is unbounded-version since a server can send an unbounded number of values to a client (i.e., $\nexists k : \forall m \in \mathcal{M} \mid |dec(m)| < k$). Note that we do not claim that `ubvStore` is a practical algorithm. We use `ubvStore` to generalize our impossibility result (Theorem 5).

Overview. We base `ubvStore` on three ideas: (i) servers assign version numbers to values, (ii) servers and clients store locally the write history (i.e., sequence of writes performed by a client) of each client and exchange write histories whenever they communicate, and (iii) a

read-only transaction by a client c is performed on c 's local knowledge of the write histories. We describe in detail these three ideas below.

An object o is served by a single server s ($o \in s.obj$). Therefore, server s can *order* the writes issued by clients to an object o by assigning incrementally increasing *version numbers* to values written to an object o . We say that a value v written to an object o , has version number vn , if v was the vn -th value written to object o . For example, if two writes to object o with values v_1 and v_2 are performed by two different clients c_{w_1} and c_{w_2} respectively, server s can assign version number 1 to value v_1 and version number 2 to value v_2 . We consider that the initial value \perp of each object has version number 0.

Both servers and clients in `ubvStore` contain local information about the write history of each client. The write history of a client c corresponds to the ordered list of the write operations client c has performed. Specifically, the write history of a client is a list of triples, where each triple is of the format $(object, value, version\ number)$. We denote with $hist_p[c]$ the write history of client c as it is known to process p . Naturally, note that process p might have stale information on the write history of client c and hence $hist_p[c]$ is a subset of client's c actual write history. For example, in a system with one server s and two clients c_1 and c_2 , server s might store locally $hist_s[c_1] = (o_1, v_1, 3) \cdot (o_2, v_3, 1)$ and $hist_s[c_2] = \epsilon$. This means that s is aware that client c_1 first performed the write of value v_1 to object o_1 and then the write of value v_3 to object o_2 , and since $hist_s[c_2] = \epsilon$, this means that s has no information about the write history of client c_2 (potentially client c_2 has not performed any writes). Similarly, client c_1 might locally contain $hist_{c_1}[c_2] = (o_5, v_9, 6)$.

Whenever a client c performs a write operation to an object served by a server s , c sends all its write histories ($hist_c$) to server s . Similarly, when a server s responds to the read request of a client c , s sends all its write histories ($hist_s$) to client c . This way, whenever servers and clients communicate, they can potentially *extend* their local write histories. Note that a server stores only values (i.e., versions) of objects that it serves. Nevertheless, a server s can store the write histories of all the clients, and which objects these clients wrote, even though s might not serve these objects. For example, a server s with $o_1 \notin s.obj$ could contain $hist_s[c_3] = (o_1, _, 9)$ where $_$ implies that s does not “know,” and hence does not store the value of object o_1 since s does not serve o_1 .

In `ubvStore`, transactions are 1-round and non-blocking. This means that when a client c performs a transaction, c sends messages to the servers serving the transaction's objects and the servers respond immediately to c . Then, client c retrieves the responses from the servers and potentially extends $hist_c$. Afterwards, c attempts to read the latest (i.e., by looking at the version number) values of each object it wants to read without violating monotonic writes. If it can read the latest values without violating monotonic writes, then client c reads these latest values. If not, client c attempts to read potentially earlier values of objects until it finds a tuple

of values that satisfies monotonic writes.

To give an example of how ubvStore performs a transaction, consider execution α_2 in Figure 4.2. In execution α_2 , client c_r sends messages to both s_1 and s_2 as part of transaction t . Server s_1 receives the message after value $v_{o_1}^1$ is visible and before value $v_{o_1}^2$ is written. Therefore, s_1 sends value $v_{o_1}^1$ to c_r . Server s_2 receives c_r 's message after value $v_{o_2}^2$ is visible and hence s_2 sends value $v_{o_2}^2$ to c_r . The issue with execution α_2 is that values $(v_{o_1}^1, v_{o_2}^2)$ cannot be read for t since these values violate monotonic writes. However, since we consider an unbounded-version data store, server s_2 can also send value $v_{o_2}^1$ to c_r . In ubvStore, when client c_r receives the message from server s_1 , c_r has $hist_{c_r}[c_w] = (o_1, v_{o_1}^1, 1)$.

When c_r receives the message from s_2 , c_r knows that $hist_{c_r}[c_w] = (o_1, v_{o_1}^1, 1) \cdot (o_2, v_{o_2}^1, 1) \cdot (o_1, \perp, 2) \cdot (o_1, v_{o_1}^2, 2)$. Knowing $hist_{c_r}$, client c_r can safely read values $(v_{o_1}^1, v_{o_2}^1)$ for transaction t .

ubvStore in detail. The algorithm of ubvStore is presented in algorithms 10 and 11. Lines 1-106 correspond to the code for the client and lines 107-118 correspond to the server code. Note however that the server code uses the EXTEND and subsequently the mergeClientHist procedure. ubvStore uses both procedures, as well as event-based (i.e., **trigger** and **upon event**) techniques [33]. A trigger event corresponds to the transmission of a message, while an upon event corresponds to the receipt of a message. A client cl has the following local variables (lines 2-4): $hist_{cl}$, $responses$, and $versionNumber$. Client cl stores the write histories in $hist_{cl}$, that is an array of lists, where each list (except $hist_{cl}[c_{init}]$ – see below) is initially empty (ϵ). For a list $list$, we denote with $|list|$ its length. We consider that there is some initial client $c_{init} \notin \mathcal{C}$ that wrote value \perp with version number 0 to all the objects. This way, if a client c performs a transaction on an object o that has not been written, the initial value \perp is read.

Algorithm 10 ubvStore for a client cl

```

1:  $\triangleright$  Client's  $cl$  local variables
2:  $hist_{cl}[c] \leftarrow \epsilon \ \forall c \in \mathcal{C}$ 
3:  $hist_{cl}[c_{init}] \leftarrow (o_1, \perp, 0) \cdot (o_2, \perp, 0) \cdots (o_k, \perp, 0)$ 
4:  $responses \leftarrow \epsilon$ ,  $versionNumber \leftarrow -1$ 
5:
6:  $\triangleright$  client  $cl$  to read objects in  $O_s$ 
7: procedure read( $O_s, t, d$ )
8:   for server  $s$  in {server that serves an object  $o \in O_s$ } do
9:      $\triangleright$  ask server  $s$  only of objects in  $O_s$  that  $s$  serves
10:    trigger  $\langle s, \text{READ} \mid O_s \cap \text{objectsServedBy}(s), t, d \rangle$ 
11:    wait until  $|responses| = |servers|$ 
12:     $hist_{cl} \leftarrow \text{extend}(hist_{cl}, responses)$ 
13:
14:     $verToRead[o] \leftarrow 0 \ \forall o \in O_s$ 
15:    for object  $o$  in  $O_s$  do
16:       $verToRead[o] \leftarrow \text{maxVerNumber}(hist_{cl}, o)$ 
17:    while true do
18:       $result \leftarrow \text{getValues}(hist_{cl}, verToRead)$ 
19:       $(safe, prObj) \leftarrow \text{isSafe}(hist_{cl}, result)$ 
20:      if  $safe$  then
21:        return  $result$ 
22:      else
23:         $verToRead[prObj] \leftarrow verToRead[prObj] - 1$ 
24:     $responses \leftarrow \epsilon$ 
25:
26:  $\triangleright$  client  $cl$  to write value  $v$  to object  $o$ 
27: procedure write( $o, v, d$ )
28:    $s \leftarrow$  server that serves  $o$ 
29:    $histToSend \leftarrow \text{clean}(hist_{cl}, s)$ 
30:   trigger  $\langle s, \text{WRITE} \mid o, v, histToSend \rangle$ 
31:   wait until  $versionNumber \neq -1$ 
32:    $hist_{cl}[c] \leftarrow hist_{cl}[c] \cdot (o, v, versionNumber)$ 
33:    $versionNumber \leftarrow -1$ 
34:   return  $ack$ 
35:
36: upon event  $\langle cl, \text{READRESPONSE} \mid r \rangle$  do
37:    $responses \leftarrow responses \cdot r$ 
38:
39: upon event  $\langle cl, \text{WRITERESPONSE} \mid vn \rangle$  do
40:    $versionNumber \leftarrow vn$ 

```

```

41: procedure isSafe( $hist_{cl}, result$ )
42:   if  $\exists o : o.val = \_ \wedge o \in result$  then
43:     return (false,  $o$ )
44:   for client  $c$  in  $\mathcal{C}$  do
45:      $\triangleright$  for triple  $t \leftarrow (val, o, vn)$ ,  $v(tr)$  corresponds to  $val$  and  $obj(tr)$  to  $o$ 
46:     if  $\exists t_1, t_2, t_3 \in hist_{cl}[c] : v(t_1), v(t_3) \in result$  then
47:       if  $r_1 \rightarrow t_2 \wedge t_2 \rightarrow t_3 \wedge obj(t_1) = obj(t_2)$  then
48:         return (false,  $obj(t_3)$ )
49:   return true
50:
51:  $\triangleright$  returns the extension of  $hist$  when utilizing  $responses$ 
52: procedure extend( $hist, responses$ )
53:    $histRes[c] \leftarrow \epsilon \ \forall c \in \mathcal{C}$ 
54:   for response  $r$  in  $responses$  do
55:     for client  $c$  in  $\mathcal{C}$  do
56:        $histRes[c] \leftarrow mergeClientHist(histRes[c], r[c])$ 
57:        $histRes[c] \leftarrow mergeClientHist(histRes[c], hist[c])$ 
58:   return  $histRes$ 
59:
60: procedure getValues( $hist_{cl}, verToRead$ )
61:   for object  $o$  in  $verToRead$  do
62:      $result \leftarrow \epsilon$ 
63:      $vn \leftarrow verToRead[o]$ 
64:      $val \leftarrow getVersion(hist_{cl}, o, vn)$ 
65:      $result \leftarrow result \cdot (o, val, vn)$ 
66:   return  $result$ 

```

```

67: procedure mergeClientHist(aHist, bHist)
68:   if size(aHist) < size(bHist) then
69:     return mergeClientHist(bHist, aHist)
70:
71:   ▷ it is guaranteed that size(aHist) ≥ size(bHist)
72:   mergedHist ←  $\epsilon$ 
73:   for triple ← (o, v, vn) in bHist do
74:     if v ≠  $\_$  then
75:       mergedHist ← mergedHist · triple
76:     else
77:       if (o, v', vn) ∈ aHist with v' ≠  $\_$  then
78:         mergedHist ← mergedHist · (o, v', vn)
79:       else
80:         mergedHist ← mergedHist · (o,  $\_$ , vn)
81:   append remaining elements of aHist to mergedHist

```

82: ▷ removes values in *history* of objects that *s* does not serve

```

83: procedure clean(history, server)
84:   histToReturn ←  $\epsilon$ 
85:   for triple ← (o, v, vn) in history do
86:     if o ∈ objectsServedBy(server) then
87:       histToReturn ← histToReturn · triple
88:     else
89:       histToReturn ← histToReturn · (o,  $\_$ , vn)
90:   return histToReturn

```

91:

92: ▷ returns the *vn*-th version value for object *o* in history *hist*

```

93: procedure getVersion(hist, o, vn)
94:   for client c in  $\mathcal{C}$  do
95:     for triple ← (o', v, vn') in hist[c] do
96:       if vn' = vn and o' = o then
97:         return v

```

98:

99: ▷ returns the greatest version number for object *o* found in history *hist*

```

100: procedure maxVerNumber(hist, o)
101:   maxvn ← 0
102:   for client c in  $\mathcal{C}$  do
103:     for triple ← (o', v, vn) in hist[c] do
104:       if maxvn < vn and o' = o then
105:         maxvn ← vn
106:   return maxvn

```

Algorithm 11 ubvStore for a server s

```

107:  $\triangleright$  Server's  $s$  local variables
108:  $hist_s[c] \leftarrow \epsilon \ \forall c \in \mathcal{C}$ 
109:  $mem[o] \leftarrow (\perp, 0) \ \forall o$  served by  $s$ 
110:
111: upon event  $\langle s, \text{READ} \mid O_s, t, d \rangle$  do
112:   trigger  $\langle cl, \text{READRESPONSE} \mid hist_s \rangle$ 
113:
114: upon event  $\langle s, \text{WRITE} \mid o, v, hist_{cl} \rangle$  do
115:    $mem[o] \leftarrow (v, mem[o].ver + 1)$ 
116:    $hist_s[c] \leftarrow hist_s[c] \cdot (o, v, mem[o].ver)$ 
117:    $hist_s \leftarrow \text{extend}(hist_s, hist_{cl})$ 
118:   trigger  $\langle cl, \text{WRITERESPONSE} \mid mem[o].ver \rangle$ 

```

Variables *responses* and *versionNumber* are used in the READ and WRITE procedures in order to inform the client that a message has been received from the server. Specifically, *responses* contains the messages received from servers during a read transaction. *versionNumber* contains the version number the server assigns to the object the client writes.

Client cl performs a transaction by calling the read procedure (lines 7-24) and providing as parameters the objects O_s , a transactional identifier t , and an additional message d . For every server s that serves an object $o \in O_s$, client cl triggers a server READ event (lines 8-10). Note that client cl only “asks” from a server s the objects that s serves using the objectsServedBy procedure. We assume that objectServedBy returns the set of all the objects served by s and this information is stored locally in cl (i.e., no communication with the server is needed). The client then waits until it receives messages from all the servers it sent a message to (Line 11). For this, client cl uses the *responses* variable that is initially ϵ and each time cl receives a response (READRESPONSE) from the server, the response is appended to *responses* (lines 36-37). When cl receives messages from all the servers it has communicated with, then $|responses|$ is equal to $|servers|$ (Line 11). After receiving the responses from the servers, client cl calls the extend procedure (Line 12) that extends $hist_{cl}$ if cl received potentially additional information from some of the servers (e.g., received information such that cl can extend a write history $hist_{cl}[c]$ for some client c). The extend procedure (lines 52-58) takes two write histories and combines them to get the maximum possible write histories using mergeClientHist as a helper procedure (lines 67-81). Afterwards, client cl stores to the *verToRead* array all the versions that it intends to read from this transaction. Initially, cl intends to get the latest version of each object (lines 15-16) and cl is able to retrieve the latest version number (as known by cl) of each object using the maxVerNumber procedure. The maxVerNumber (lines 100-106) procedure accepts write histories (*hist*) and an object o and finds the maximum version number of object o by utilizing the information in *hist*.

Client cl then enters the **loop** (lines 17-23), where cl attempts to read the latest values that satisfy monotonic writes, until it succeeds. It does so by calling the `getValues` procedure (lines 60-66). The `getValues` procedure loops through the objects to be read and finds the desired version for each object using the `getVersion` procedure. The `getVersion` (lines 93-97) procedure loops through all the write histories (similar to the `maxVerNumber` procedure) to find the value of object o with the specific version number.

Then, the client verifies that the read values are safe (i.e., satisfy monotonic writes) by calling `isSafe` (Line 19). The `isSafe` procedure (lines 41-49) takes as input the write histories ($clientHist$) and the read values ($result$) and examines whether the read values by cl violate monotonic writes (Line 47). Note that in `isSafe` client cl might get back `_` as the value of an object (Line 42), meaning that cl knows that a value was written by some client but is not aware of this value. In such a case, client cl attempts to read the immediately previous value of that object in the next loop, by decrementing the version that is about to be read for that object (Line 23) and repeating the loop. At the end, the client sets $responses$ back to ϵ (Line 24). Specifically, `isSafe` checks whether there exist 3 triples (i.e., 3 writes) t_1 , t_2 , and t_3 performed by the same client, where the first write takes place before the second write ($t_1 \rightarrow t_2$) and the second write before the third write ($t_2 \rightarrow t_3$). Furthermore, `isSafe` checks whether the two values $v(t_1)$ and $v(t_3)$ that have been read by the client, and value $v(t_1)$ of $obj(t_1)$ has been overwritten before the write of value $v(t_3)$. If this is the case, the values violate monotonic writes and `isSafe` returns $obj(t_3)$ (Line 48). Note that the monotonic writes property is violated because the client attempted to read the version of $obj(t_3)$, and therefore we call $obj(t_3)$ the problematic object. If the read values are safe (Line 49), the read values are returned (Line 21). Otherwise, the version needed for object $prObj$ is decremented by one (Line 23) and the loop repeats.

To perform a write operation, client cl calls the `write` procedure (lines 27-34). Client cl retrieves the server that serves object o (Line 28), cleans history $hist_{cl}$ by removing values of objects server s does not serve (Line 29) and then cl sends a `WRITE` message to the server (Line 30). Recall that a server does not store values of objects that it does not serve. Therefore the client calls the `clean` procedure (lines 83-90) that simply goes through all the triples in the provided history and if the history of some client contains a value that is not served by the server it stores `_` as the value. Afterwards, the client waits until the server responds (Line 31) (i.e., $versionNumber$ is updated when receiving a `WRITERESPONSE` from the server (lines 39-40). Finally, client cl updates its write histories from the one received by the client (Line 32), sets $versionNumber$ back to -1 and returns an acknowledgment.

The code for a server s is presented in Algorithm 11. As with a client, server s stores locally all the write histories ($hist_s$). Additionally, s stores the values of the objects it serves in the mem array (Line 109). For each object $o \in s.obj$, $mem[o]$ corresponds to a pair with the latest written value and its corresponding version number. When server s receives a `READ` event, it responds

to the client with a READRESPONSE that contains $hist_s$ (Line 112). Note that server s does not update its state in response to a *read* event, and hence reads in `ubvStore` are invisible. When server s receives a WRITE event to write a value v to object o (Line 114), s stores the value in $mem[o]$ and increases its version number by one (Line 115). Then, server s (potentially) extends its current knowledge of $hist_s$ by the one received ($hist_{cl}$) from the client using the `extend` procedure (Line 117). Finally, server s responds to the client with a WRITERESPONSE including a version number (Line 118).

The cautious reader might notice that some parameters (e.g., d) are not used. Nevertheless, such parameters appear in `ubvStore` to adhere to the model we present in Section 4.2.

Correctness. To prove that `ubvStore` is correct, we have to show that `ubvStore`: (i) is a valid data store (i.e., all its executions are well-formed), (ii) provides monotonic writes, and (iii) provides minimal progress.

Valid data store. Naturally each per-process execution is well-formed since processes follow `ubvStore`'s algorithm. Any execution has correct issues, since clients send requests for objects to servers that serve them (Line 10). A server only responds to a client after receiving a request and hence any execution has valid responses. Additionally, clients are sequential if we force them to wait for an operation's response before invoking a new operation. Furthermore, any execution has distinct values and no-transaction reuse; we assume that clients append their client identifier and a constantly increasing counter to the values they write, as well as the transactional identifiers they use. Any execution generated by `ubvStore` has valid values since a client c only reads values contained in $hist_c$. The values contained in $hist_c$ are received by some server and hence were written; values are only appended to a client history by the server (Line 116). Finally, we assume that any execution messages are not received out of thin air and that there is no message duplication. We can achieve this by implementing well-known techniques [33]. We do not incorporate these techniques in `ubvStore` to avoid making `ubvStore` verbose.

Monotonic writes. `ubvStore` trivially provides monotonic writes. A client returns values from a read-only transaction only if these values are safe (Line 49 in Algorithm 10). The fact that values are safe means that there is no monotonic writes violation (Line 47) and the `isSafe` procedure is successful. However, we still have to prove the following lemma to show that `ubvStore` does not attempt to read versions that do not exist.

Lemma 24. *For every $read(O_s, t, d)$ performed by a client c , for every object $o \in O_s$, it is the case that $verToRead[o] \geq 0$.*

Proof. Someone might think that $verToRead[o] < 0$ due to Line 23. We show that this is not the

case. Assume by way of contradiction that during a $\text{read}(O_s, t, d)$ operation by some client c , there is an object $o \in O_s$ such that $\text{verToRead}[o] < 0$. For this to happen, it should be the case that $\text{verToRead}[o] = 0$ and a subsequent isSafe call (Line 19) returns (false, o) . This would imply that $\text{verToRead}[o]$ gets decremented by one and hence $\text{verToRead}[o] < 0$. However, we show that if $\text{verToRead}[o] = 0$, the isSafe cannot return o as a problematic object. Since for o to be returned as a problematic object the following needs to happen. When isSafe is called, there exist a client c' such that there exist triples $t_1, t_2, t_3 \in \text{clientHist}[c']$ such that $t_1 \rightarrow t_2 \wedge t_2 \rightarrow t_3$, $\text{obj}(t_1) = \text{obj}(t_2)$ and $\text{obj}(t_3) = o$. The fact that $t_2 \rightarrow t_3$ implies that client c' first performed a write to $\text{obj}(t_2)$ and later to object $\text{obj}(t_3) = o$. However, if $\text{verToRead}[o] = 0$ this means that the read value is the initial value, namely \perp , and t_3 should have been some initial write which is not the case. A contradiction. Note that we also need to consider the case where isSafe returns (false, o) for object o after successfully executing the if-statement in Line 42. This is not possible because all the clients have the value \perp for version number 0, and hence if $\text{verToRead}[o] = 0$, then $o.\text{val} = \perp \neq _$. \square

Minimal progress. To show that ubvStore provides minimal progress we have to show that ubvStore is eventually responsive and bounded visible. ubvStore is eventually responsive since by construction the servers respond to the client's requests (i.e., reads or writes). ubvStore is also bounded visible. To see this, consider a finite execution $\alpha \in \mathcal{J}$ which last event e_w is a client write request that writes value v to object o served by server s , as in Definition 17. Furthermore, consider all $r > 0$ completed client write requests to o before e_w that write values, where all the written values are contained in the set v_{old} . Then for $b = 0$, for any extension $\alpha' \in \mathcal{J}$ of α , every transaction that appears for the first time after $|a| + b$ in α' and that requests o does not read a value that belongs to $v_{old} \cup \{\perp\}$. By way of contradiction, consider a transaction t that appears for the first time after $|a| + b$ and that reads o and that reads a value that belongs to $v_{old} \cup \{\perp\}$. When transaction t contacts server s , server s sends back at least the latest value v of object o . For transaction t to read a value that belongs to $v_{old} \cup \{\perp\}$, it must be the case that isSafe returns (false, o) from Line 48. Note that isSafe cannot return (false, o) from Line 42 because server s returns value $v \neq _$ for object o . Because transaction t started its execution after the write of v (due to a third write t_3 as seen in Line 47), transaction t also sees the write of t_2 and t_1 . Therefore, object o cannot be the problematic object. A contradiction.

As a final note, we devised ubvStore to clarify and generalize our theoretical impossibility result (Theorem 5). In practice, sending several versions in one round is *costly*. Therefore, we expect that an implementation of ubvStore would perform worse than COPS-SNOW [87]. However, this is not an apples to apples comparison, because COPS-SNOW can violate consistency during an asynchronous period, while ubvStore never violates consistency.

4.5 Conclusion

Our framework is inspired by the models of Attiya et al. [14] and Burckhardt et al. [32]. However, in addition, our framework captures transactions. As far as we know, our formal framework is the first to precisely capture the notion of fast transactions, and specifically the notion of bounded-version data stores. We formally defined bounded-version data stores by introducing the decoding function *dec* and the definition of valid values, something we believe is a contribution in itself. In contrast, other models [45, 71, 87] that attempt to define fast transactions, fail to precisely define the restrictions imposed by one-version reads (i.e., servers could potentially “cheat” and respond with more than one version).

Lu et al. [87] introduce the informal notion of latency-optimal read-only transactions (i.e., fast reads in Definition 12), as well as present the SNOW theorem. The SNOW theorem states that we cannot devise a read-only transaction algorithm that satisfies the following four properties: (i) strict serializability, (ii) non-blocking reads, (iii) one-round and one-version reads, and (iv) coexistence with write transactions. Konwar et al. [71] revisit the SNOW theorem and present some new results that consider the role of client-to-client messaging to the SNOW theorem. Tomsic et al. [118] investigate the relation between the consistency, the speed, and the freshness of reads and among others show that visible fast transactions are possible by reading an arbitrarily old snapshot of the database and hence such a data store does not provide the weak property of bounded visibility. Didona et al. [44, 45] look into causally-consistent data stores and investigate the cost fast read-only transactions impose on writes, as well as prove that a data store cannot provide fast read-only transactions in combination with write transactions. In contrast to previous work [44, 45, 71, 87, 118], we consider the weaker consistency model of monotonic writes and hence we strengthen our impossibility result. Finally, note that real systems [5, 8, 40] that provide fast transactions, either assume strong synchrony (e.g., Spanner [40] relies on global time), or provide visible read-only transactions and hence are not fault-tolerant.

Finally, to the best of our knowledge, our work is the first to bring fault-tolerance and fast transactions together into attention. As a matter of fact, the original work of Lu et al. [87] that informally introduces fast transactions, presents the COPS-SNOW data store that supports such fast transactions. COPS-SNOW is able to keep operating during a network partition and hence the loss of one or more data centers. However, COPS-SNOW cannot tolerate the failure of even a single server, since otherwise consistency is violated.

In this chapter, we proved that invisible fast transactions are impossible in a data store that supports the weak consistency model of monotonic writes. To prove our result, we devised a formalization to precisely capture notions such as one-round, one-version, etc. Our proof is the first one to shed light on an important and unexplored consequence of fast transactions, that is, a data store that supports fast transactions cannot tolerate the failure of even one server.

Additionally, with `ubvStore`, we showed that the number of versions a server can send to a client is consequential to whether transactions can be both non-blocking and 1-round trip.

5 Concluding Remarks

In this thesis, we revealed hidden complexities (in the sense of intricacies or costs) of distributed systems that challenge accepted truths in the field of distributed computing. In what follows, we summarize the main findings of this thesis and propose avenues for future research.

Consensus. In Chapter 2 we defined what it means for a consensus algorithm to be leaderless. To the best of our knowledge, this work is the first to formally introduce the notion of a leaderless algorithm, initiating the study of provable leaderless consensus algorithms. Then, we devised and proved correct Archipelago, a leaderless consensus algorithm for the shared-memory model, and subsequently translated our result to the message-passing model. Additionally, we conjectured that Archipelago could be transformed to tolerate Byzantine failures. It would be interesting to see if this conjecture holds and how such a Byzantine-tolerant algorithm performs in practice.

State Machine Replication. In Chapter 3, we examined the relation between consensus and State Machine Replication (SMR) in terms of their complexity. We proved the surprising result that SMR is more expensive than a repetition of consensus instances. Concretely, we showed that in a synchronous system where a single instance of consensus always terminates in a constant number of rounds, completing one SMR command can potentially require a non-constant number of rounds. Additionally, we supported our formal proof with experimental results using two well-known SMR implementations (a Multi-Paxos and a Raft implementation). An interesting direction for future work would be to optimize SMR performance for the worst-case, for example by expediting recovery [95], perhaps with applicability in performance-sensitive applications. Another possible direction would be to investigate whether it is possible to design SMR algorithms where replicas balance among themselves the burden of keeping each other up to date collaboratively (e.g., as attempted in [21]).

Transactions. In Chapter 4, we proved that fast transactions are impossible. Specifically, we proved that invisible fast transactions are impossible in a data store that supports the weak consistency model of monotonic writes. Our proof sheds light on an important and unexplored consequence of fast transactions, that is, a data store that supports fast transactions cannot tolerate the failure of even one server. In this regard, this thesis brings a better understanding of the relationship between fault-tolerance and fast transactions. A possible direction for future research is to investigate whether a different reasonable definition of fast transactions exists that allows for practical fault-tolerant data stores.

Bibliography

- [1] Amazon EC2. <http://aws.amazon.com/ec2/>. [Online; accessed 8-July-2020].
- [2] etcd. <https://github.com/coreos/etcd>. [Online; accessed 8-July-2020].
- [3] LibPaxos3. <https://bitbucket.org/sciascid/libpaxos>. [Online; accessed 8-July-2020].
- [4] MongoDB. <https://www.mongodb.com>. [Online; accessed 8-July-2020].
- [5] MySQL Cluster. <https://www.mysql.com/products/cluster/>. [Online; accessed 8-July-2020].
- [6] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin. Revisiting fast practical Byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017.
- [7] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC*, 2004.
- [8] M. K. Aguilera, J. B. Leners, and M. Walfish. Yesquel: Scalable sql storage for web applications. In *SOSP*, 2015.
- [9] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. M. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *ICDCS*, 2016.
- [10] K. Antoniadis, R. Guerraoui, D. Malkhi, and D.-A. Seredinschi. State machine replication is more expensive than consensus. In *DISC*, 2018.
- [11] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *DSN*, 2017.
- [12] J. Aspnes, H. Attiya, and K. Censor. Max registers, counters, and monotone circuits. In *PODC*, 2009.
- [13] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *JACM*, 42(1):124–142, 1995.

- [14] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. *TPDS*, 28(1):141–155, 2017.
- [15] P. Aublin, S. B. Mokhtar, and V. Quéma. RBFT: redundant byzantine fault tolerance. In *ICDCS*, 2013.
- [16] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. *TOCS*, 32(4):12:1–12:45, Jan. 2015.
- [17] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. In *VLDB*, 2013.
- [18] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, 2013.
- [19] P. Bailis and K. Kingsbury. The network is reliable. *ACM Queue*, 12(7):20, 2014.
- [20] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC*, 1983.
- [21] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *ATC*, 2013.
- [22] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with bft-smart. In *DSN*, 2014.
- [23] M. Biely, P. Robinson, U. Schmid, M. Schwarz, and K. Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks. *TCS*, 726:41 – 77, 2018.
- [24] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, 1993.
- [25] E. Borowsky and E. Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *PODC*, 1997.
- [26] Z. Bouzid, A. Mostfaoui, and M. Raynal. Minimal synchrony for Byzantine consensus. In *PODC*, 2015.
- [27] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC*, 2000.
- [28] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook’s distributed data store for the social graph. In *ATC*, 2013.
- [29] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *PDP*, 2004.
- [30] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

-
- [31] Y. Buchnik and R. Friedman. Fireledger: A high throughput blockchain consensus protocol. *VLDB*, 13(9):1525–1539, 2020.
 - [32] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.
 - [33] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.
 - [34] D. Cason, P. J. Marandi, L. E. Buzato, and F. Pedone. Chasing the tail of atomic broadcast protocols. In *SRDS*, 2015.
 - [35] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *TOCS*, 20(4):398–461, 2002.
 - [36] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, 2007.
 - [37] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, 43(2):225–267, 1996.
 - [38] B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, Apr 2009.
 - [39] G. Chockler, R. Guerraoui, and I. Keidar. Amnesic distributed storage. In *DISC*, 2007.
 - [40] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google’s globally distributed database. *TOCS*, 31(3):8:1–8:22, 2013.
 - [41] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. DBFT: Efficient leaderless Byzantine consensus and its application to blockchains. In *NCA*, 2018.
 - [42] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.
 - [43] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. In *SOSP*, 2007.
 - [44] D. Didona, P. Fatourou, R. Guerraoui, J. Wang, and W. Zwaenepoel. Distributed transactional systems cannot be fast. In *SPAA*, 2019.
 - [45] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel. Causal consistency and latency optimality: friend or foe? In *VLDB*, 2018.
 - [46] E. W. Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

Bibliography

- [47] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *SoCC*, 2014.
- [48] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, Apr. 1988.
- [49] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155 – 173, 1982.
- [50] C. Fernández-Campusano, M. Larrea, R. Cortiñas, and M. Raynal. Eventual leader election despite crash-recovery and omission failures. In *PRDC*, 2015.
- [51] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.
- [52] E. Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *PODC*, 1998.
- [53] Á. García-Pérez, A. Gotsman, Y. Meshman, and I. Sergey. Paxos consensus, deconstructed and abstracted (extended version). *CoRR*, abs/1802.05969, 2018.
- [54] C. Georgiou, S. Gilbert, R. Guerraoui, and D. R. Kowalski. On the complexity of asynchronous gossip. In *PODC*, 2008.
- [55] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.
- [56] S. Gilbert and N. Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
- [57] V. Gramoli, L. Bass, A. Fekete, and D. Sun. Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations. *TPDS*, 27(9):2711–2724, Sep 2016.
- [58] R. Guerraoui. Indulgent algorithms (preliminary version). In *PODC*, 2000.
- [59] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental consistency guarantees for replicated objects. In *OSDI*, 2016.
- [60] D. Gupta, L. Perronne, and S. Bouchenak. BFT-Bench: Towards a practical evaluation of robustness and effectiveness of BFT protocols. In *DAIS*, 2016.
- [61] M. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, Jan. 1991.
- [62] H. Howard and J. Crowcroft. Coracle: Evaluating Consensus at the Internet Edge. In *SIGCOMM*, 2015.
- [63] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *OPDIS*, 2016.
- [64] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *ATC*, 2010.

-
- [65] P. R. Johnson and R. Thomas. Maintenance of duplicate databases. RFC 677, 1975.
 - [66] ZooKeeper documentation. <https://zookeeper.apache.org/doc/r3.6.1/>. [Online; accessed 8-July-2020].
 - [67] F. Junqueira and B. Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, Inc., 2013.
 - [68] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: Preliminary version. *SIGACT News*, 32(2):45–63, June 2001.
 - [69] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.
 - [70] R. Klophaus. Riak core: Building distributed applications without shared state. In *CUFP*, 2010.
 - [71] K. M. Konwar, W. Lloyd, H. Lu, and N. A. Lynch. The SNOW theorem revisited. *CoRR*, abs/1811.10577, 2018.
 - [72] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.
 - [73] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *EuroSys*, 2013.
 - [74] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
 - [75] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.
 - [76] L. Lamport. The weak Byzantine generals problem. *JACM*, 30(3):668–676, July 1983.
 - [77] L. Lamport. The part-time parliament. *TOCS*, 16(2):133–169, 1998.
 - [78] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
 - [79] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, pages 22–23. Springer, 2003.
 - [80] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
 - [81] L. Lamport. Leaderless Byzantine consensus, 2010. United States Patent, Microsoft, Redmond, WA (USA).
 - [82] L. Lamport. Brief announcement: Leaderless Byzantine Paxos. In *DISC*, 2011.
 - [83] L. Lamport, D. Malkhi, and L. Zhou. Stoppable Paxos. *TechReport, Microsoft Research*, 2008.

Bibliography

- [84] L. Lamport and M. Massa. Cheap paxos. In *DSN*, 2004.
- [85] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *SOSP*, 1991.
- [86] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *SOSP*, 2011.
- [87] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *OSDI*, 2016.
- [88] S. Lukasik. Why the Arpanet was built. *IEEE Annals of the History of Computing*, 2011.
- [89] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.
- [90] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report TR-11-21, The University of Texas at Austin, 2011.
- [91] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *OSDI*, 2008.
- [92] P. J. Marandi, S. Benz, F. Pedone, and K. Birman. Practical experience report: The performance of paxos in the cloud. *CoRR*, 2014.
- [93] C. Martín, M. Larrea, and E. Jiménez. Implementing the omega failure detector in the crash-recovery failure model. *Journal of Computer and System Sciences*, 75(3):178–189, 2009.
- [94] S. A. Mehdi, C. Little, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *NSDI*, 2017.
- [95] O. M. Mendizabal, F. L. Dotti, and F. Pedone. High performance recovery for parallel state machine replication. In *ICDCS*, 2017.
- [96] I. Moraru, D. G. Andersen, and M. Kaminsky. There is More Consensus in Egalitarian Parliaments. In *SOSP*, 2013.
- [97] Y. Moses and S. Rajsbaum. A layered analysis of consensus. *SICOMP*, 31(4):989–1021, 2002.
- [98] A. Mostéfaoui, H. Moumen, and M. Raynal. Signature-free asynchronous binary byzantine consensus with $t < n/3$, $o(n^2)$ messages, and $o(1)$ expected time. *JACM*, 62(4):1–21, 2015.
- [99] A. Mostefaoui and M. Raynal. Low cost consensus-based atomic broadcast. In *Proceedings. 2000 Pacific Rim International Symposium on Dependable Computing*, 2000.

-
- [100] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, 2014.
 - [101] M. Oriol, M. Wahler, R. Steiger, S. Stoeter, E. Vardar, H. Koziolk, and A. Kumar. Fasa: A scalable software framework for distributed control systems. In *ISARCS*, 2012.
 - [102] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *TOCS*, 33(3):7:1–7:55, Aug. 2015.
 - [103] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *JACM*, 27(2):228–234, Apr. 1980.
 - [104] M. O. Rabin. Randomized Byzantine generals. In *FOCS*, 1983.
 - [105] N. Santoro and P. Widmayer. Time is not a healer. In *STACS*, 1989.
 - [106] N. Santoro and P. Widmayer. Agreement in synchronous networks with ubiquitous faults. *TCS*, 384(2):232 – 249, 2007.
 - [107] N. Santos and A. Schiper. Tuning paxos for high-throughput with batching and pipelining. In *ICDCN*, 2012.
 - [108] U. Schmid, B. Weiss, and I. Keidar. Impossibility results and lower bounds for consensus under link failures. *SICOMP*, 38(5):1912–1951, 2009.
 - [109] U. Schmid, B. Weiss, and J. Rushby. Formally verified byzantine agreement in presence of link faults. In *ICDCS*, 2002.
 - [110] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.
 - [111] E. Sit, A. Haeberlen, F. Dabek, B.-G. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and J. Kubiatowicz. Proactive Replication for Data Durability. In *IPTPS*, 2006.
 - [112] K. Spirovska, D. Didona, and W. Zwaenepoel. Optimistic causal consistency for geo-replicated key-value stores. In *ICDCS*, 2017.
 - [113] K. Spirovska, D. Didona, and W. Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *DSN*, 2018.
 - [114] K. Spirovska, D. Didona, and W. Zwaenepoel. Paris: Causally consistent transactions with non-blocking reads and partial replication. In *ICDCS*, 2019.
 - [115] P. Sutra. On the correctness of egalitarian Paxos. *CoRR*, abs/1906.10917, 2019.
 - [116] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

Bibliography

- [117] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *TODS*, 4(2):180–209, 1979.
- [118] A. Z. Tomsic, M. Bravo, and M. Shapiro. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Middleware*, 2018.
- [119] M. van Steen and A. S. Tanenbaum. *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017.
- [120] G. M. D. Vieira, I. C. Garcia, and L. E. Buzato. Seamless paxos coordinators. *CoRR*, abs/1710.07845, 2017.
- [121] G. Voron and V. Gramoli. Dispel: Byzantine SMR with distributed pipelining. *CoRR*, abs/1912.10367, 2019.
- [122] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.
- [123] B. Wester, J. A. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, 2009.
- [124] Y. C. Yeh. Safety critical avionics for the 777 primary flight controls system. In *DASC*, 2001.
- [125] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, 2019.
- [126] K. Zuse. *The Computer-My Life*. Springer Science & Business Media, 1993.

Karolos Antoniadis

CONTACT	Phone: +41 (0) 76 70 23 10 6 Email: karolos.antoniadis@epfl.ch
LANGUAGES	English: TOEFL iBT Test (2012, score: 107/120) German: Mittelstufe Goethe-Institut (2007, C1) Greek: Mother tongue
EDUCATION	Doctoral Student in Computer and Communication Sciences École Polytechnique Fédérale de Lausanne (EPFL) Distributed Computing Laboratory September 2016 – (expected) August 2020 Master’s Degree in Computer Science Eidgenössische Technische Hochschule (ETH) Zürich September 2012 – November 2015 (GPA 5.24/6) Degree (Ptychio) in Computer Science University of Crete, Greece September 2007 – July 2012 (GPA 9.44/10, summa cum laude)
WORK EXPERIENCE	Software Engineer Intern, Twitter, Seattle June – September 2019: Member of the Coordination team working on ZooKeeper. Among others, implemented a Jepsen-like tool for injecting faults in a ZooKeeper cluster. Identified and fixed bugs in ZooKeeper’s leader election algorithm. Agile Software Engineer Intern, eBay, Zurich March – September 2014: Member of a Scrum team developing tools for real-time monitoring of eBay’s retail campaigns. Coded and tested back-end components written in Java.
PUBLICATIONS	Leaderless Consensus , co-authored with Vincent Gramoli, Rachid Guerraoui, Eric Ruppert, and Igor Zablotchi. (under submission) Thread-Placement Learning , co-authored with Rachid Guerraoui, and Vasileios Trigonakis. (to appear) (<i>40th International Conference on Distributed Computing Systems (ICDCS)</i> , 2020) The Impossibility of Fast Transactions , co-authored with Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel. (<i>34th International Parallel and Distributed Processing Symposium (IPDPS)</i> , 2020) [Book Chapter] The Notions of Time and Global State in a Distributed System , co-authored with Rachid Guerraoui in “Concurrency: The Works of Leslie Lamport.”, 2019 In personal communication, Turing Award Laureate Leslie Lamport said: <div style="padding-left: 40px;">I was asked to check the page proofs of the ACM book about my work [...] You gave a very nice presentation of the work and said more about its influence than I would have been able to. And yours was the one chapter in which I didn’t find even the tiniest error—a testimony to the effort you must have put into it.</div>

State Machine Replication is More Expensive than Consensus, co-authored with Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. (*32nd International Symposium on Distributed Computing (DISC)*, 2018)

The entropy of a distributed computation random number generation from memory interleaving, co-authored with Peva Blanchard, Rachid Guerraoui, and Julien Stainer. (*Distributed Computing Journal, Volume 31*, 2018)

Sequential Proximity - Towards Provably Scalable Concurrent Search Algorithms, co-authored with Rachid Guerraoui, Julien Stainer, and Vasileios Trigonakis. (*5th International Conference on Networked Systems (NETYS)*, 2017)

TEACHING ASSISTANT EXPERIENCE **Concurrent Algorithms, EPFL** (Fall 2018, 2019)
Software Engineering, EPFL (Fall 2017)
Practice of Object-Oriented Programming, EPFL (Spring 2017, 2018, 2019)
Probability II, University of Crete (Spring 2009)

ADDITIONAL RESEARCH EXPERIENCE **Research Intern at EPFL School of Computer and Communication Sciences Distributed Computing Laboratory**
January – June 2016: Worked on a random number generator that leverages the unpredictability of memory interleaving in modern processors.

May – November 2015: Finished my master thesis titled “Sequential Proximity - Towards Provably Scalable Concurrent Search Algorithms,” in which I formalized properties that lead to scalable concurrent search data structures.

Undergraduate Research Fellow at Foundation for Research & Technology - Hellas (FORTH), Institute of Computer Science
Computer Architecture and VLSI Systems Laboratory
October 2011 – July 2012: Finished my diploma thesis titled “On Multi-Version Software Transactional Memories” which surveyed some of the most important multi-version software transactional memory algorithms.

Information Systems Laboratory
July – September 2009: Developed an application for generating RDF Schemas.

HONORS AND AWARDS Computer and Communication Sciences Fellowship at EPFL (2016) for outstanding academic achievements

Distinguished undergraduate scholarship “Stelios Orphanoudakis” from FORTH (years: 2009, 2010)

Scholarship from State Scholarships Foundation (IKY) for being the first in my class the first, second and third year of my studies (years: 2008, 2009, 2010)

Honors prize award, “Thales” competition, 2007, hosted by the Greek Mathematical Society

Honors prize award, “Second Phase” (2007) and “First Phase” (2006) Informatics Competition (PDP) hosted by the Greek Computer Society

PROGRAMMING LANGUAGES (in order of experience) Java, C, and Python