Time-Efficient Asynchronous Service Replication

Vom Fachbereich Informatik der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades eines Doktor-Ingenieur (Dr.-Ing.)
vorgelegt von

Dipl.-Inform. Dan Dobre

aus Lugoj, Rumänien

Referenten:
Prof. Neeraj Suri
Prof. Michel Raynal

Datum der Einreichung: 23.06.2010
Datum der mündlichen Prüfung: 30.09.2010

Darmstadt 2010
D17

Abstract

Modern critical computer applications often require continuous and correct operation despite the failure of critical system components. In a distributed system, fault-tolerance can be achieved by creating multiple copies of the functionality and placing them at different processes. The core constitutes a distributed protocol run among the processes whose goal is to provide the end user with the illusion of sequentially accessing a single correct copy. Not surprisingly, the efficiency of the distributed protocol used has a severe impact on the application performance.

This thesis investigates the cost associated with implementing fundamental abstractions constituting the core of service replication in asynchronous distributed systems, namely (a) *consensus* and (b) the *read/write register*. The main question addressed by this thesis is how efficient implementations of these abstractions can be. The focus of the thesis lies on *time complexity* (or *latency*) as the main efficiency metric, expressed as the number of communication steps carried out by the algorithm before it terminates. Besides latency, important cost factors are the resilience of an algorithm (i.e. the fraction of failures tolerated) and its message complexity (the number of messages exchanged).

Consensus is perhaps the most fundamental problem in distributed computing. In the consensus problem, processes propose values and unanimously agree on one of the proposed values. In a purely asynchronous system, in which there is no upper bound on message transmission delays, consensus is impossible if a single process may crash. In practice however, systems are not asynchronous. They are timely in the common case and exhibit asynchronous behavior only occasionally. This observation has led to the concept of unreliable failure detectors to capture the synchrony conditions sufficient to solve consensus.

This thesis studies the consensus problem in asynchronous systems in which processes may fail by crashing, enriched with unreliable failure detectors. It determines how quickly consensus can be solved in the common case, characterized by stable executions in which all failures have reliably been detected, settling important questions about consensus time complexity.

Besides consensus, the read/write register abstraction is essential to sharing information in distributed systems, also referred to as *distributed storage* for its importance as a building-block in practical distributed storage and file systems. We study fault-tolerant read/write register implementations in which the data shared by a set of clients is replicated on a set of storage base objects. We consider *robust* storage implementations characterized by (a) wait-freedom (i.e. the fact the read/write operations invoked by correct clients always return) and (b) strong consistency guarantees despite a threshold of object failures. We allow for the most general type of object failure, Byzantine, without assuming authenticated data to limit the adversary. In this model, we determine the worst-case time complexity of accessing such a robust storage, closing several fundamental complexity gaps.

Kurzfassung

Für moderne sicherheitskritische Computeranwendungen ist eine ununterbrochene und fehlerfreie Funktion erforderlich, oft auch dem Ausfall kritischer Systemkomponenten zum Trotz. In einem verteilten System kann Fehlertoleranz dadurch erreicht werden, dass mehrere identische Kopien einer Applikation erstellt, und auf verschiedene, möglicherweise fehleranfällige Prozesse plaziert werden. Kern dieses Verfahrens ist ein verteiltes Protokoll, das von den Prozessen im verteilten System ausgeführt wird, mit dem Ziel eine einzelne und ausfallsichere Kopie zu simulieren. Endbenutzern wird der Eindruck vermittelt, auf eine korrekte, hochverfügbare Kopie sequentiell zuzugreifen. Wie nicht anders zu erwarten hat die Effizienz des verwendeten, verteilten Protokolls eine signifikante Auswirkung auf die Performanz der Applikation.

Diese Dissertation untersucht die Kosten grundlegender Abstraktionen verteilten Rechnens, die den Kern der Replikation von Diensten in verteilten Systemen bilden, nämlich (a) Consensus und (b) das Lese-/Schreibregister. Die Hauptfragestellung dieser Arbeit ist wie effizient Implementierungen dieser Abstraktionen überhaupt sein können. Dabei liegt das Augenmerk der Dissertation auf der *Zeitkomplexität* (oder *Latenzzeit*) als maßgebliche Effizienzmetrik, gegeben durch die Anzahl der Kommunikationsphasen (oder -schritte) die ein verteiltes Protokoll benötigt bevor es terminieren kann. Zwei wichtige Kostenfaktoren neben der Latenzzeit sind die Ausfallsicherheit (die Anzahl der tolerierten Ausfälle) und die Nachrichtenkomplexität (die Anzahl der gesendeten Nachrichen) eines Protokolls.

Consensus ist höchstwahrscheinlich das grundlegendste Problem auf dem Gebiet des verteilten Rechnens. Es kann wie folgt beschrieben werden: Prozesse schlagen jeweils einen Wert vor und müssen sich auf einen der vorgeschlagenen Werte einigen. In einem rein asynchronen System, in dem keine oberen Schranken für die Kommunikationszeit zwischen Prozessen existieren, ist Consensus unlösbar, selbst wenn nur ein einziger Prozess ausfallen darf. In der praktischen Anwendung sind allerdings solche Systeme meistens synchron (d.h. es gibt solche oberen Schranken), und sie verhalten sich nur gelegentlich asynchron. Diese Beobachtung führte zu dem Konzept des unverlässlichen Fehlerdetektors, der die hinreichenden Synchronitätsbedingungen für die Lösbarkeit von Consensus erfasst und abstrahiert.

Diese Arbeit untersucht das Consensus-Problem in asynchronen Systemen mit Anhalte-Ausfällen von Prozessen, und Verfügbarkeit von Fehlerdetektoren, die auch unverlässliche Angaben über den Fehlerzustand von Prozessen machen dürfen. Es wird ermittelt wie schnell Consensus in Fällen die in der Praxis häufig auftreten gelöst werden kann in, z.B. in sogenannten stabilen Ausführungsinstanzen, in denen geschehene Ausfälle bereits verlässlich erkannt worden sind, und keine weiteren Ausfälle mehr stattfinden. Offene Fragen nach der Latenzzeit von Consensus werden durch die Ergebnisse dieser Arbeit geklärt.

Neben Consensus, ist auch das Lese- und Schreibregister eine grundlegende

Abstraktion auf dem Gebiet des verteilten Rechnens, und ermöglicht den Prozessen in einem verteilten System auf gemeinsame Daten zuzugreifen. Das Lese- und Schreibregister wird oft auch, wegen seiner Relevanz als Baustein in praktischen verteilten Speicher- und Dateisystemen, als *Storage* bezeichnet.

Diese Dissertation erforscht fehlertolerante Storage-Implementierungen, in denen Daten, die von Clients gemeinsam genutzt werden, aus Gründen der Verlässlichkeit und Hochverfügbarkeit auf mehrere Storage-Server repliziert und damit redundant gespeichert werden.

Es werden *robuste* Storage-Implementierungen betrachtet, die sich (a) durch Wartefreiheit (d.h. von korrekten Clients aufgerufene Lese- und Schreiboperationen müssen stets terminieren) und (b) durch starke Konsistenzeigenschaften auszeichnen, trotz der Fehlfunktion von Storage-Servern und Clients. Das untersuchte Systemmodell erlaubt die allgemeinste Klasse von Funktionsfehlern, sogenannte Byzantinische Fehler, ohne eine Authentifizierung der Daten anzunehmen um den Angreifer zu begrenzen. In diesem Rahmen wird die Worst-Case Latenzzeit von Lese- und Schreibzugriffen auf ein robustes Storage untersucht und ermittelt, und dadurch werden etliche grundlegende Komplexitätslücken geschlossen.

Acknowledgements

To begin with, I am deeply grateful to my advisor Prof. Neeraj Suri for the huge amount of confidence he has placed in me, and for the unlimited freedom I received for developing my own ideas and research direction. To the thesis committee for the time spent reading and evaluating my thesis. Special thanks goes to my co-advisor Prof. Michel Raynal who's scientific writings have opened an entire new chapter in my professional life.

A big thanks goes to my close colleagues and dear friends, Marco and Matthias, from the distributed computing "subgroup" at DEEDS. They always have offered their availability and devoted their time and attention to endless discussions about various research topics, and also about personal subjects. Special thanks goes to Marco for helping me to improve most of my writings, for the good time we had in Darmstadt, and the weeks spent together on our fun travels to foreign countries. Also, special thanks to Matthias who always took the time to listen to my ideas, pointing out the bad and the boring ones, since the first day he joined. I thank all my coauthor colleagues for their commitment to our joint publications, and the more senior ones for their guidance.

I am grateful to my colleagues and friends Dinu, Marco, Matthias, Peter and Piotr with whom I've spent reams of pleasant lunch breaks and "Stammtisch" evenings. To all DEEDS members, for making my stay in the group an extremely enjoyable experience. To our secretary Sabine for taking care of the paper work I never had to handle, and to our technical assistant Ute, who always provided me with a flawless machine setup.

I am deeply grateful to my parents, for always caring, for all their love and unconditional support. I'm dedicating this thesis to my wife Alina, for her love and understanding, and for the most beautiful present — the baby we are eagerly awaiting.

Dan
Darmstadt, October 4, 2010

Preface

This thesis concerns the Ph.D. work I did under the supervision of Prof. Neeraj Suri at the Computer Science Department, Technische Universität Darmstadt, from 2004 to 2010. The thesis focuses on time-efficient asynchronous distributed algorithms and lower bounds in the context of (a) consensus and state machine replication resilient to crash failures, and (b) distributed storage resilient to Byzantine failures. This work is a composition of four published papers [DS06, DMS08, DMSS09, DMSS10], as well one paper that has been submitted for publication to peer reviewed conferences/journals [DGM+10].

[DGM+10] Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, and Marko Vukolic. The Complexity of Robust Atomic Storage. 2010. Technical Report TR-TUD-DEEDS-06-01-2010.

[DMS08] Dan Dobre, Matthias Majuntke, and Neeraj Suri. On the Time-complexity of Robust and Amnesic Storage. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 197–216, 2008.

[DMSS09] Dan Dobre, Matthias Majuntke, Marco Serafini, and Neeraj Suri. Efficient Robust Storage Using Secret Tokens. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 269–283, 2009.

[DMSS10] Dan Dobre, Matthias Majuntke, Marco Serafini, and Neeraj Suri. HP: Hybrid Paxos for WANs. In *EDCC'10: Proceedings of the 8th European Dependable Computing Conference*, pages 117–126, 2010.

[DS06] Dan Dobre and Neeraj Suri. One-step Consensus with Zero-Degradation. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 137–146, 2006.

During this period, besides the work presented in the thesis, I also worked on (1) Byzantine resilient atomic broadcast and state machine replication algorithms [DRS07, SBD+10], (2) on abortable fork-linearizable storage [MDSS09] and (3) on eventually linearizable concurrent objects [SBD+10].

[DRS07] Dan Dobre, HariGovind V. Ramasamy, and Neeraj Suri. On the Latency Efficiency of Message-Parsimonious Asynchronous Atomic Broadcast. In *SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 311–322, 2007.

[MDSS09] Matthias Majuntke, Dan Dobre, Marco Serafini, and Neeraj Suri. Abortable Fork-Linearizable Storage. In *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*, pages 255–269, 2009.

[SBD+10] Marco Serafini, Peter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas. In *DSN '10: Proceedings of the 40th International Conference on Dependable Systems and Networks*, 2010. To Appear.

[SDM+10] Marco Serafini, Dan Dobre, Matthias Majuntke, Peter Bokor, and Neeraj Suri. Eventually Linearizable Shared Objects. In *PODC '10: Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, 2010. To Appear.

Contents

Chapter 1

Introduction

A core engineering principle when building safety-critical systems is to avoid a single point of failure. By relying on the correct operation of individual components, the failure of a single component may result in the unavailability, or even worse, the corruption of the entire system. A widely used approach for solving this problem is to design the system in a redundant way, by using replication. This is also true for modern critical computer applications that cannot afford data loss or data corruption resulting from failures. This thesis is about efficiently implementing reliable computer systems from unreliable components, by means of replication in distributed systems.

A distributed system consists of a set of computing entities, also called processes, which are able to perform local computation and to communicate with each other. Distributed computing encompasses the study of fundamental problems and algorithms in distributed systems.

The main two challenges distributed computing is facing are *failures* and *asynchrony*. With the advent of cheaper storage, computing and communication resources, distributed systems have been increasingly built to support massive scalability in clusters often consisting of thousands of commodity servers [GGL03, DHJ+07]. In order to meet the needs of high throughput and low latency, business critical data is partitioned and processed in parallel on multiple machines. In such large-scale systems, it has been recognized that failures are commonplace rather than being exceptions [GGL03]. Some failures are accidental, and can be detected (e.g. by using cross check sums) and semantically turned into simple *crash* failures. However, when corporate networks are exposed to the internet, the provided service can be compromised by malicious intruders resulting in undetectable *arbitrary* behavior of the faulty components, called *Byzantine* failures [PSL80].

Besides failures, *asynchrony* poses a considerable challenge to distributed computing. In asynchronous systems there are no bounds on transmission

delays nor on processor speed, making it impossible to distinguish between a crashed process and a very slow one. Since processes can be arbitrarily fast or arbitrarily slow, the correctness of a solution cannot rely on the timely delivery of messages from processes. It would seem easier to design algorithms in a model which assumes bounds on processing and communication delays (i.e. the *synchronous* model). In such a model, unresponsive processes can be easily detected using end-to-end timeouts. However, modern applications are composed of many layers, each with complex timing assumptions and thus they cannot always guarantee end-to-end timing properties. At best, these systems have predictable response time in the common case, but even a slight deviation of the load or the operating conditions can lead to long delays which may violate the timing assumptions made. Additionally, when dealing with open networks, such as the Internet, malicious break-ins by attackers may target the timely delivery of messages in order to compromise the service.

1.1 Context

Above we have given an overview of the main challenges that need to be addressed by fault-tolerant asynchronous distributed computing research. This thesis concentrates on two fundamental abstractions in distributed computing, namely *consensus* [LSP82, FLP85, DLS88, CT96, Lam98] and *read/write storage* (equivalently read/write register) [Lam86, ABD95, MR98, JCT98].

1.1.1 Consensus

Consensus is perhaps the most fundamental and mostly studied problem in distributed computing. It has been introduced by Pease, Shostak and Lamport [PSL80]. In the consensus problem, processes propose values and are required to irrevocably agree on a value such that: (a) no two processes decide differently (Agreement), (b) eventually every correct process decides (Termination) and (c) if a process decides a value v, then some process has proposed v (Validity) [CT96].

Consensus is an essential building block for many critical applications. For instance, the most popular way to maintain application consistency and availability in the presence of failures (and asynchrony) is state machine replication [Lam78, Sch90]. A reliable server is emulated by a collection of unreliable replica servers, some of which may fail, and replicas *agree* on a sequence of requests to be executed. With this approach, all replicas perform operations that update the data in the same order, and thus remain mutually

consistent. Agreement on a sequence of requests boils down to running a sequence of consensus instances, one per client request (or group of client requests).

Another example for the relevance of consensus are distributed transactions [Gra78, GL06], where processes need to agree whether to commit or to abort a transaction. Generally speaking, consensus is *universal*, meaning that the problem of implementing *any* type of shared object can be reduced to solving consensus [Her91].

Despite its importance as a distributed computing abstraction, deterministic consensus has no solution in the asynchronous model if a single process may crash [FLP85]. However, real systems are not completely asynchronous. As a consequence, a great number of works have explored ways to circumvent this impossibility [BO83, DDS87, DLS88, CT96, CF98, MRR03].

One way of solving consensus is by extending the asynchronous model with timing assumptions about message transmission times. The *eventually synchronous* model [DLS88] assumes the existence of an *unknown* upper bound on transmission time. The bound is not even required to hold a priory. However, there is a time called global stabilization time (GST), such that after GST this bound on message transmission time holds.

A particularly interesting and widely adopted approach constitutes the seminal concept of *unreliable failure detectors* and their classification [CT96]. Since the impossibility of consensus in the asynchronous model stems from the inherent difficulty to tell a crashed process from a very slow one, the idea was to extend the asynchronous model with failure detection capabilities. Local failure detector modules monitor a subset of processes and output information about suspected processes. After a finite time (e.g. GST), a failure detector is required to cease making mistakes. For instance, faulty processes eventually must not be mistakenly taken for correct and vice versa. Obviously, failure detectors cannot be implemented in a purely asynchronous model. Their role is merely to encapsulate sophisticated timing assumption and to abstract the synchrony requirements sufficient for solving consensus.

One failure detector type which is of particular interest is the eventual *leader oracle*, Ω, that eventually outputs the same correct leader process at all processes. Ω has been shown to be the weakest failure detector for solving consensus [CHT96], and many consensus algorithms have been devised for this model, e.g. [DG02, GR04, Lam98].

In this thesis we study the consensus problem in the asynchronous system model with crash failures, enhanced with failure detectors. Under the notion of asynchronous consensus we consider *indulgent* [Gue00] algorithms. An algorithm is indulgent if it *always* preserves its *safety* properties (e.g. agreement) even in asynchronous executions, and ensures termination in execu-

tions in which failure detection eventually is reliable. The price payed for the indulgence is that no more than a *minority* of processes can be faulty [CT96], no matter what (unreliable) failure detector is used.

Besides consensus, the thesis investigates the equivalent problem of *atomic broadcast* [CT96]. Atomic broadcast constitutes the core of state machine replication. It ensures that requests broadcast by clients to a group of replica servers are delivered to all servers in the group in the same order. Given that atomic broadcast is typically built from consensus (e.g. [CT96]), its performance is determined by the consensus algorithm used.

1.1.2 Distributed Storage

Although the problem of implementing a reliable service from unreliable components involves some form of agreement, not all reliable service implementations translate to consensus. An example of such a service is the read/write register abstraction, for its relevance in practical distributed storage and file system architectures also called *read/write storage*. It provides two primitives, a *write* operation which writes a value into the register, and a *read* operation which returns a value previously written. The read/write register abstraction is essential to sharing information in distributed systems because it abstracts away the complexity incurred by concurrent access to shared data. Besides its API being very simple, it is today the heart of modern "cloud" key-value storage APIs (e.g. Amazon S3 [AWS]).

Distributed storage algorithms constitute an active area of research and are appealing alternatives to centralized storage systems based on specialized hardware [AEMCC+05, CDH+06, ASV06, SFV+04]. Typically, a reliable read/write storage is implemented by replicating the data on a set of fault-prone *base objects*, of which a threshold may fail. The clients access the base objects over which the storage is implemented, and the end user is provided with the illusion of accessing a centralized storage.

Read/write storage can be classified according to the consistency semantics it provides and the cardinality of readers and writers it supports [Lam86, AW98]. This thesis concentrates on a fundamental class of read/write storage, in which there are multiple readers and a single writer (MRSW) [ABD95, ACKM06, ACKM07, GV06, GV07]. Standard transformations known in the literature can be applied to implement a multi-writer storage from a single-writer one [AW98].

Also, read/write storage comes in three consistency flavors *safe*, *regular* and *atomic* in increasing strength [Lam86]. A safe storage guarantees that a read which does not overlap with any write returns the last value written. However, if a read is concurrent with a write, the read may return an ar-

bitrary value, which clearly limits the applicability of safe storage. Regular storage strengthens safety, requiring that a read returns an actually written value that is not older than the last value written. This makes regular storage appealing as a direct building block for other applications (e.g. shared memory consensus [ACKM06]). However, the most desirable consistency criterion is *atomicity* (also called *linearizability* [HW90]). Atomicity provides to the clients accessing the storage (possibly in a concurrent manner) the illusion that data is accessed sequentially.

In this thesis we focus on distributed storage in the *arbitrary* failure model (also called *Byzantine*), which becomes increasingly relevant in absence of the full trust in the cloud [CKS09]. In this model, we study distributed storage that provides strong consistency guarantees (i.e., regularity or atomicity) and wait-freedom [Her91], (i.e., the fact that read/write operations invoked by correct clients always eventually return) despite (a) asynchrony and the failure (possibly Byzantine) of any number of clients and (b) the largest possible number of Byzantine base object failures.

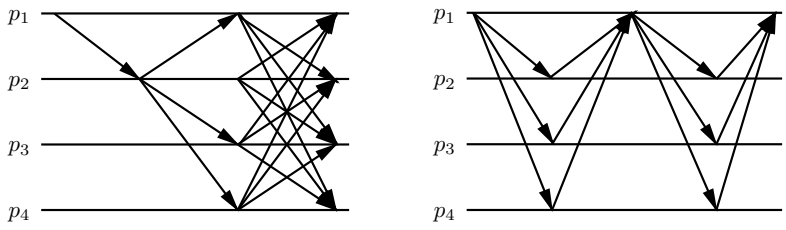
1.1.3 On Time Complexity and Related Metrics

Two of the most important challenges when devising a distributed algorithm is (a) to tolerate the largest possible number of faults, called *optimal resilience* and (b) to provide optimal efficiency with respect to some relevant complexity metric. An essential efficiency measure of distributed algorithms is their *time complexity*, (also called *latency*). Roughly speaking, latency captures how quickly a given algorithm can terminate. Time complexity is typically measured as the number of *message delays* (or *steps* of communication) an algorithm takes before it terminates [Awe85, Sch97, AW98] (Figure 1.1 (a)). In data centric storage, often communication takes place only between clients and servers (e.g. when servers are active disks). Then, the latency of an algorithm is measured as the number of communication *round-trips* (or simply *rounds*) [ACKM06, GV06], where one round is equivalent to two message delays (illustrated in Figure 1.1 (b)).

The focus of the this thesis lies on designing latency efficient algorithms. Besides being of great theoretical importance, the exploration of the latency metric extends beyond the associated intellectual challenge. With the growth in data processing and storage outsourcing driven by the advent of cloud computing, the number of remote interactions among processes maps to our latency metric and is often directly associated with the monetary cost. This obviously increases the practical relevance of devising algorithms which are latency efficient.

Two other relevant efficiency metrics considered in the thesis are *through-*

put and *message complexity*. Throughput is measured as the number of requests that can be handled per time unit [GGL03, vRS04, MJM08] and message complexity [RC05, GGK07] is the total number of messages exchanged.



(a) 3 communication steps (b) 2 rounds (4 steps)

Figure 1.1: Time complexity (Latency)

1.2 Motivation

1.2.1 Consensus Time Complexity and Open Questions

Given that the synchrony models that we have discussed allow completely asynchronous executions which are finite but unbounded, it is generally impossible to bound the running time of consensus in the worst case. In real systems however, there are often long periods during which communication is timely, i.e., many executions are actually synchronous. In such executions, failure detection based on time-outs can be reliable. Therefore, the performance of asynchronous consensus is studied in synchronous executions, or equivalently in executions in which failure detectors are accurate.

Two major research trends have emerged in investigating the latency of consensus. A large amount of effort has been devoted to executions which are synchronous from the outset, with and without failures [MR01, KR01, DG02, GR04, DG05]. More recently, it has been studied how long it takes for consensus to recover from arbitrary periods of asynchrony once the system becomes synchronous and no more failures occur [DGK07, KS06]. The latter is relevant for understanding consensus performance in systems that frequently oscillate between periods of stability and instability.

Keidar and Rajsbaum [KR01] established a tight bound on the latency of consensus of *two* communication steps before global decision is reached, (i.e., before all correct processes decide) in *nice* executions which are failure-free and synchronous. This result contrasts with the synchronous model, in which

global decision is reached after *one* step in failure-free executions, pointing out the inherent cost associated with tolerating asynchrony.

In practice, nice executions are the exception rather than the norm, as observed in [GGL03]. Hence, the question arises if the optimal latency of two communication steps can also be attained in executions with failures, e.g. in which all failures have occurred *before* the execution starts? Recall that, state machine replication involves executing many instances of consensus, and therefore, it is important that failures which have occurred earlier do not affect the performance of later instances. Executions in which failure detection is reliable and all failures are initial are termed *stable*, and algorithms that attain the optimal latency in stable executions are called *zero-degrading* [DG02, GR04].

A large amount of work has went into circumventing the above two-step lower bound, and several papers have been published devising asynchronous consensus algorithms which, for certain vectors of input values (also called *configuration*), expedite global decision to *one* communication step, e.g. [BGMR01, PSUC02, PS03, Lam06a].

Special attention is paid to the case when all processes propose the same value, which is particularly relevant in the context of state machine replication, e.g. in datacenters [PS03]. To see why, it is important to understand that atomic broadcast, which lies at the heart of state machine replication, typically consists of two phases, a *broadcast* phase followed by a consensus phase [CT96]. In the broadcast phase, clients send their requests to the server processes. When a server process receives a request, it triggers a new consensus instance proposing that particular request.

In many networks, such as LANs, it often happens that requests broadcast by different clients are received by all servers in the same order, a phenomenon called *spontaneous total order* [PS03]. Thus, when a new consensus instance is triggered, all server processes propose the same value in that particular instance. Optimizing consensus in this regard expedites atomic broadcast from *three* message delays in the common case to just *two*.

The original one-step consensus algorithm is due to Brasileiro et al. [BGMR01]. The algorithm attains global decision after a single all-to-all message exchange if all proposals are equal; otherwise it falls back to a generic consensus algorithm. While being very efficient from a configuration where all proposals are the same, the algorithm requires at least *three* communication steps from other configurations. A closer look at other algorithms that reach consensus in one communication step [PS03, MR00, PSUC02, CMP06, Lam06a] reveals that they also fail to match the two-step lower bound of Keidar and Rajsbaum [KR01].

A natural question to ask is whether a single algorithm exists that matches

both lower bounds. More specifically, a number of intriguing questions arise:

(Q1.1) Does a single consensus algorithm exist that attains global decision (a) in one communication step when all proposals are equal and (b) in two communication steps in stable runs?

(Q1.2) What failure detector is sufficient to attain these two properties? Is Ω, which is sufficient for (b) also sufficient for both (a) and (b)?

(Q1.3) If Ω is insufficient, then what are possible ways to circumvent the impossibility? Does it help to employ a stronger failure detector, or even to weaken the problem in a meaningful way?

In arbitrary networks, e.g. a WAN, the assumption that messages broadcast from clients to servers experience a spontaneous total order is too optimistic [SPMO02]. When different clients broadcast their requests roughly at the same time, it often happens that they are received in different orders by the replicas, which is termed a *collision*. Thus, ways have been investigated to minimize the impact of collisions on consensus performance, for instance by relaxing the assumptions under which consensus ca be expedited [PS02, Lam05, Zie05].

Pedone and Schiper [PS02] point out that more efficient algorithms can be devised by taking into account the "semantics" of messages. Instead of blindly totally ordering all the messages, the authors propose to totally order only *conflicting* messages, according to a binary conflict relation defined on the messages. The practical relevance is obvious. For instance, real-world applications often encounter read-dominated workloads and "read" operations never conflict with each other, so they can be applied in any order. In contrast, "writes" conflict with other operations applied to the same object, and consequently have to be totally ordered. The authors of [PS02] introduce the *generic broadcast* problem in which only the conflicting messages are totally ordered, and describe an algorithm that attains the optimal latency of *two* message delays with non-conflicting messages.

Inspired by this work, Lamport [Lam05] introduces a very clean and precise generalization of consensus, from agreement on a single value, to agreement on a growing partially ordered set of values (called *generalized consensus*). The algorithm that solves the problem is a variation of the well-known Fast Paxos protocol [Lam06a], called Generalized Paxos, featuring optimal message complexity and optimal resilience (i.e., $2f+1$ servers, where f is the bound on faults). It requires the optimal *two* message delays when requests are non-conflicting (including the additional broadcast step from clients to

servers). However, it incurs *four* additional message delays to recover from collisions caused by conflicting requests.

Zielinski's generic broadcast [Zie05] effectively eliminates collision recovery, by running multiple protocols in parallel and choosing the quickest outcome. Although the implementation is latency optimal, the authors acknowledge that it is prohibitively expensive in terms of message and computation complexity. Taking a practical perspective, we raise the following question:

(Q2) Is it possible to devise a high throughput and low latency algorithm that shares all the nice features of Generalized Paxos (i.e., optimal messages and optimal resilience) without the expense of collision recovery?

1.2.2 Storage Time Complexity and Open Questions

We now turn our attention to the second subject of the thesis, namely the read/write register abstraction. Several papers have explored the solvability and the time complexity metric in the context of a read/write register.

The Crash-failure Model

A seminal crash-tolerant and wait-free atomic MRSW register implementation with optimal resilience (i.e., $2t + 1$ processes, where t is the bound on faults) was presented in [ABD95]. As it constitutes a key paradigm for distributed storage design, we briefly discuss its main ideas.

In [ABD95], each process assumes both the roles of client and base object and up to a minority of processes may crash. Every write operation completes in a single round. The writer holds a monotonically increasing *timestamp*, which induces a total order on the values written, corresponding to the real-time order of write operations. A write operation assigns a fresh timestamp to the value it writes, and broadcasts a message containing the timestamp-value pair to all processes. Each process locally stores the value with the highest timestamp received so far. After receiving a higher timestamped value, each process stores the timestamp-value pair and acknowledges the receipt. The write operation completes when it collects acknowledgments from a majority of processes.

Beside writes, *regular* reads in [ABD95] also complete in a single round. The reader broadcasts a message to all processes requesting the timestamp-value pair stored on each of them. A process that receives the message simply replies with the timestamp-value pair it has locally stored. After receiving timestamp-value pairs from a majority of processes, the read completes by returning the value with the highest timestamp.

It is not difficult to see that the implementation is wait-free, i.e., that read/write operations always return. Regularity is ensured by the intersection property of majority sets (also called *quorums*). If a write operation with timestamp ts has completed, then a quorum is updated with ts and the corresponding value. A subsequent read operation accessing a quorum, reads from at least one of the processes updated by the write. Thus, a read never returns a value with a timestamp lower than ts.

Atomic reads in [ABD95] require one additional *write back* phase, whose purpose is for the reader to update a quorum with the timestamp-value it is going to read. This ensures that if a read returns a value with timestamp ts, then no subsequent read returns a value with a lower timestamp. Atomicity comes at the expense of two communication rounds for read operations.

The problem of modifying [ABD95] to enable single round reads was explored in [DGLC04], which showed that such *fast* atomic implementations, (i.e. every operation completes in one round) are possible, albeit they come with the price of limited number of readers and suboptimal resilience. Moreover, the reader in [DGLC04] needs to write (i.e., modify the objects' state) as dictated by the lower bound of [FL03] which showed that every atomic read must write into at least t objects. The limitation on the number of readers of [DGLC04], was relaxed in [GNS09], where a crash-tolerant MRSW atomic register implementation was presented, in which most of the reads complete in a single round, yet a fraction of reads is permitted to be slow and complete in two rounds.

The Byzantine-failure Model

The study of reliable distributed storage initiated in [ABD95] for the crash model was extended to the Byzantine model in [MR98, JCT98], in which (a) any number of clients may crash and (b) a threshold of base objects may manifest arbitrary failures. An essential difference to the crash model is that any safe storage implementation tolerating t Byzantine faults requires at least $3t + 1$ base objects (optimal resilience) [MAD02]. To see why, note that any two quorums must overlap in $t + 1$ objects to ensure that some non-malicious object is contained in the intersection.

In the Byzantine setting, several different data and communication models have been explored. Some works assume a model where data is authenticated (called self-verifying data) [MR98, CT06, DGLV05], typically using digital signatures. The time complexity of these algorithms is in line with that of crash-tolerant distributed storage protocols, e.g. [ABD95, DGLC04]. On the downside, they involve a certification and a key pre-distribution phase and entail a noticeable computation overhead. Also, they are typically based

on unproven cryptographic assumptions and they are not secure against computationally unbounded adversaries.

Thus, a great number of works have tackled the problem of Byzantine-fault tolerant storage in a model in which data is *unauthenticated* [MAD02, BD04, GWGR04, AAB07, ACKM06, ACKM07, GV06, GLV06, GV07, CGK07, HGR07]. Research in the unauthenticated model comes in two different flavors, according to the power of the base objects assumed.

In the *server centric model* base objects are *active*, characterized by the ability to push messages to subscribed clients and to communicate with other base objects, e.g. [MAD02, BD04, AAB07]. Protocols in this model however, do not scale well with the number of clients and base object faults, due to their high message complexities. A different flavor is the *data centric model*, in which objects are *passive*. Passive objects only reply in response to client requests and do not communicate with other base objects. Algorithms designed for this model are more general, because little is assumed about passive base objects.

Robust Storage

In the thesis, we focus on *robust* storage [CGK07] implemented from passive storage components. A robust storage algorithm wait-free implements (at least) a regular storage from Byzantine base objects in the unauthenticated data model.

Robust algorithms for unauthenticated data are particularly difficult to design when values previously stored are not permanently kept in storage, but similar to a circular buffer, they can be overwritten by a sequence of values written after them. Obviously, this is desirable because it enforces a limited amount of data to be stored, preventing the base objects from exhausting their memory. Algorithms that satisfy this property are called *amnesic* [CGK07]. Amnesic algorithms store in the base objects only a limited, typically small history of written values (if any). Thus, the amnesic property captures an important aspect of the space requirements of a distributed storage implementation. In contrast, non-amnesic algorithms store an unlimited number of values in the base objects, e.g. [MAD02, BD04, GWGR04, GV06, GV07, AAB07].

The difficulty of implementing robust amnesic storage stems from the fact that in the unauthenticated data model, the value read must be sampled from more than one base object, to guarantee that it is not forged. When during a read operation, written values are progressively erased by a sufficient number of overlapping writes, it has been shown to be impossible for the read to complete, if readers are precluded from writing [CGK07].

Many Byzantine resilient algorithms avoid the problem of storing an unlimited number of values in the base objects by relaxing robustness. For instance, some implementations do not ensure wait-freedom [Her91] but weaker termination guarantees, such as obstruction-freedom [HGR07] introduced in [HLM03], or finite-writes [ACKM06]. Other works implement only weaker safe storage semantics [JCT98, MR98, ACKM06, GV06]. Only two works have explored the feasibility of robust and amnesic storage [GV06, ACKM07]. The algorithm presented in [GV06] is not *bounded wait-free* and reads require an unbounded number of rounds in the worst case. The one described in [ACKM07], albeit very elegant and simple, has non-optimal resilience and non-optimal time complexity. Thus, a natural question to ask is whether robust algorithms which are also amnesic are inherently more costly than non-amnesic ones. Specifically, the state of the art leaves the following questions open:

(Q3.1) What is the worst-case time complexity of robust amnesic storage? Is it possible to devise a robust and amnesic algorithm that is *fast*, i.e., where each operation completes in one round?

(Q3.2) What is the worst-case time complexity of robust amnesic storage with optimal resilience? Does a bounded wait-free algorithm exist, and if yes can it match the latency of non-amnesic storage?

Robust storage implementations for unauthenticated data are particularly attractive because they do not incur the overhead of cryptography and they are invulnerable to cryptographic attacks. However, existing unauthenticated algorithms with optimal resilience and optimal time complexity [ACKM06, GV06] have a much higher (worst-case) *read* latency compared to algorithms storing self-verifying data, using digital signatures [MR98]. This is critical because many practical workloads are dominated by read operations.

For instance, [ACKM06] studied the read/write latency of unauthenticated storage with optimal resilience, under the constraint that readers are precluded from writing. The authors of [ACKM06] showed a tight lower bound on writing of two rounds into MRSW safe storage, and a tight lower bound on reading of $t+1$ rounds from such a storage. Precluding readers from writing is appealing because it results in implementations able to support an unbounded number of possibly malicious readers with constant memory at the servers. However, allowing readers to modify the base objects' state helps improve latency as shown in [GV06], through a two-round tight lower bound on reading from optimally resilient robust MRSW regular storage.

Altogether compared to optimally resilient algorithms in the authenticated model, which feature *fast* read/write operations [MR98, CT06], these

results are rather sobering. Thus, the question arises if it is possible to achieve better latency in the unauthenticated model without fundamentally strengthening the assumptions. Specifically, we raise the following questions:

(Q4.1) Is the unauthenticated model inherently more costly in terms of read complexity compared to the authenticated model?

(Q4.2) Is there a way to circumvent the above read lower bounds without the overhead of cryptography? Specifically, can we achieve constant read complexity if readers do not write? Can we expedite reads to be fast?

Robust Atomic Storage

In the context of Byzantine-fault tolerant storage, few papers have explored the *best-case* latency of optimally resilient robust atomic storage. Here, best-case encompasses synchrony, no or few object failures and the absence of read/write concurrency. [GLV06] presented the first atomic storage implementation in which both reads and writes are fast in the best-case (i.e., complete in a single round-trip). Furthermore, [GV07] considered robust atomic storage implementations with the possibility of having fast reads and writes gracefully degrade to two or three rounds, depending on the size of the available quorum of correct objects. Surprisingly, despite the wealth of research on robust atomic storage, there is no general picture about the *worst-case*, leaving the following question open:

(Q5) What is the worst-case time complexity of robust atomic storage?

1.3 Contributions

In the previous section, we have motivated the need for further exploration of the time complexity of consensus and read/write storage. The contributions of the thesis consist of a series of results, including algorithms and lower bounds, that collectively aim at providing adequate answers to the questions raised in the sections 1.2.1 and 1.2.2. Onwards we give an overview of the results by briefly describing each of this thesis' contributions.

1.3.1 (C1) One-step Consensus with Zero-Degradation

Our first goal is to explore if there is a single consensus algorithm that is (a) one-step if all proposal values are equal and (b) matches the lower bound of two communication steps in every stable execution (i.e., is zero-degrading).

Thus, we aim at determining if one-step consensus needs at least *three* communication steps in general, answering questions **Q1.1**, **Q1.2** and **Q1.3**.

As a first contribution we show that no consensus algorithm relying on Ω can be at the same time one-step and zero-degrading. In a sense, these two properties are incompatible. To get a rough idea why, note that in a leader based consensus algorithm, a correct process decides the value proposed by the current leader (in the first communication step) after being echoed by a quorum of processes (in the second communication step). In a one-step algorithm, a correct process decides the value proposed by a quorum of processes, not necessarily including the leader process. Obviously, agreement is violated if the two decision values are different. Thus, a third step is needed to resolve the conflict.

Our second goal is to find sufficient conditions for circumventing the established impossibility and to eliminate the third step. We consider two different treatments of the problem and present corresponding consensus protocols. In the first approach, we condition one-step decision on the behavior of the failure detector. With this relaxation, one-step decision is guaranteed only in stable executions. The corresponding consensus algorithm we describe in the thesis extends beyond theoretical interest. The stability of executions mostly depends on how hard it is to implement the properties of the failure detector used (e.g. how many timely links are needed). Among the failure detectors that allow solving consensus, Ω is the easiest to implement in a real system. Since, algorithms using Ω are more likely to exhibit stable behavior, optimizing latency in this respect is appealing.

The second approach circumvents the impossibility by using the strictly stronger failure detector $\diamond\mathcal{P}$, which eventually outputs exactly the set of faulty processes. The algorithm is inspired by Lamport's work on Fast Paxos [Lam06a], and guarantees both one-step decision (irrespective of the failure detector output) and zero-degradation. However, it is important to note that $\diamond\mathcal{P}$ being strictly stronger than Ω, its properties are harder to satisfy in practice.

Furthermore, we present a consensus-based atomic broadcast algorithm that has a latency of *two* message delays in every (stable and) collision-free execution and *three* message delays in every stable execution (which is optimal). We evaluate the algorithm using our two consensus implementations in a LAN of workstations and compare them to the state of the art. The results indicate that the theoretical latency bounds are reflected only for low to moderate load. With increasing load, the frequency of collisions also increases, and the protocol mostly operates in the slower mode. Moreover, the additional messages exchanged to enforce fast message delivery, negatively affects peak throughput.

1.3.2 (C2) Generalized Consensus and Hybrid Paxos

We address the problem of degraded performance caused by frequent collisions, by turning our attention to the generalized consensus problem, recently introduced by Lamport [Lam05]. Our contribution is to devise a generalized consensus algorithm that meets all latency, message complexity and resilience lower bounds, and that provides a competitive peak throughput. Thus we answer question **Q2**, on the practicality of generalized consensus, in the affirmative. To fully appreciate our contribution, prior to describing our result, we first give a short introduction to Lamport's novel consensus framework [Lam06b] and the Paxos protocol family [Lam98, Lam06a, Lam05].

Background

In traditional state machine replication, a sequence of instances of a consensus protocol are used to agree on the sequence of client commands, where the ith consensus instance chooses the ith command. Alternatively, a *single* instance of consensus can be used to choose a *sequence* of commands. Moreover, if many commands commute, only non-commutable commands need to be ordered. Exploiting this observation, generalized consensus [Lam05] chooses a growing partially ordered set of commands, called a *history*, in which every pair of non-commutable commands is ordered.

Lamport's definition of (generalized) consensus is stated in a slightly different framework than the traditional consensus considered so far, making it directly applicable to state machine replication in a client/server environment. Specifically, Lamport [Lam03] considers three different types of roles, played by the processes: *proposers* that propose commands, *acceptors* that choose an increasing command history and *learners* that learn what history has been chosen. In a client/server architecture, clients might play the roles of proposer and learner, and servers might play the role of acceptor. In addition, a leader is elected among the acceptors to coordinate their actions. In the traditional consensus framework considered thus far, each process takes the role of proposer, acceptor and learner.

Optimal consensus protocols expressed in this framework are the well known Classic Paxos (CP) [Lam98] and the more recent Generalized Paxos (GP) [Lam05]. In stable executions, CP requires three message delays. The communication pattern during normal operation is Client → Leader → Acceptors → Learners. GP saves one message delay by having the clients send their proposals directly to the acceptors and bypassing the leader, thus requiring two message delays. Note that these latencies are identical to those of atomic broadcast. Also, they map to the optimal two message delays

(respectively one) of traditional consensus, where the first step is ignored.

GP works fine if the acceptors receive the same sequence of conflicting commands. However, when conflicting commands are received in different orders, this results in no command being chosen. To ensure progress, GP runs a collision recovery procedure, adding four extra message delays, and a significant computational and message overhead. Thus, if collisions are frequent, GP has a higher latency and a lower throughput than CP.

Also, we found that even in the absence of collisions, depending on the layout of clients and servers, CP can outperform GP (for many clients). This stems from the fact that in order to be fast, GP needs larger quorums than CP. The quorums accessed by GP are termed *fast quorums*.

Our Contribution

Our contribution is a novel generalized consensus protocol called Hybrid Paxos (HP), which provides the best features of GP and CP together. HP essentially is an extension of CP by an additional "fast mode", enabling *fast learning* in the absence of collisions. By fast learning we mean learning in two message delays like in GP. However, unlike GP, in stable executions HP takes at most three message delays, which is the best-case latency of CP. In addition to these latencies being optimal [Lam06b], they are attained with linear messages and $2f + 1$ acceptors, which is also optimal.

We show for the first time that generalized consensus can be used to build efficient replicated services in practice. The key to efficiency is that fast learning must not impact the bottleneck, which in CP is the leader. Additional messages in HP are exchanged only between clients (which are both proposers and learners) and acceptors. Thereby, HP is able to exploit the underutilization of acceptors in CP, offering a lower latency than CP up to 70% of its peak throughput. In addition, fast learning is enabled only if spare capacity is available. This is done by adaptively switching fast learning on and off based on the load. Our evaluation using Emulab [WLS+03] shows that the latency of HP indeed reaches the theoretical minimum. Also, that in the presence of collisions and with increasing load, HP behaves like CP.

1.3.3 (C3) Optimal Robust Amnesic Storage

In the context of distributed storage, we first study the read complexity of robust amnesic algorithms. The goal is to determine if robust algorithms which are also amnesic, are inherently more expensive in terms of latency than non-amnesic ones, answering questions **Q3.1** and **Q3.2**. Given that with robust amnesic storage, on each read, the reader must write into the

base objects, as dictated by the impossibility of [CGK07], one may intuitively think that there is no fast read implementation. Maybe surprisingly, we show that such fast read implementations exist, and also that reading from amnesic storage in general can be as fast as reading from non-amnesic storage.

Specifically, our contribution consists of two robust and amnesic algorithms. The first algorithm is optimal in terms of latency while the second one exhibits minimal latency combined with optimal resilience. The developed algorithms are based on a novel concurrency detection mechanism and a helping procedure, by which a writer detects overlapping reads and helps them to complete.

Our first developed algorithm is *fast*, meaning that *every* operation (read and write) completes in only *one* round of communication with the base objects. It requires $4t + 1$ base objects to tolerate t Byzantine failures. It is worthwhile noting that the combination of latency and resilience is optimal, as with fewer base objects at least *two* rounds are needed for both reads and writes to complete [ACKM06, GV06].

The second developed algorithm uses the optimal number of $3t + 1$ base objects and is the first bounded wait-free algorithm with optimal resilience. Moreover, every read operation completes in *two* communication rounds, which has been shown to be optimal [GV06]. The only other existing robust and amnesic algorithm with optimal resilience has an unbounded read latency in the worst case [GLV06].

We now briefly explain the intuition behind the approach. Our algorithms employ a novel reverse communication scheme between writer and reader, in which the reader stores information used by the writer to detect concurrent operations. This communication between reader and writer is abstracted in a separate shared object called a *safe counter* (one per reader), whose value is advanced by the reader and read by the writer. The values returned by the counter are termed *views* and each read operation is associated with a unique view. When a read operation has advanced its current view, a subsequent write operation can read the new updated view. When the writer detects a concurrent read operation rd, indicated by a view change, it *freezes* the last value v previously written. Freezing v means that v is not overwritten unless the read operation rd has completed. Basically, this scheme guarantees that rd samples $t + 1$ copies of v, which would ensure that v is not forged. We note that rd does not violate regularity by returning v. Essentially this is true because all the values written after v are written by concurrent write operations. However, to preclude that read operations return old values previously frozen, the writer assigns to each frozen value the latest view, as a freshness indicator for the reader.

1.3.4 (C4) Robust Storage using Secret Tokens

Our second contribution in the context of robust storage aims at bridging the complexity gap between robust algorithms and algorithms storing self-verifying data, answering questions **Q4.1** and **Q4.2**.

We describe two robust storage implementations for unauthenticated data with optimal resilience and optimal time complexity. The first algorithm supports unbounded readers and features constant read complexity, while the second algorithm features fast reads. Our algorithms circumvent the lower bounds established in [GV06, ACKM06] by using *secret tokens*. A secret token (briefly token) is a value randomly selected by the client and attached to the messages sent to the base objects. The secrecy property of a token selected by a correct client is that the adversary can not generate its value before the client actually uses the token.

Secret tokens are useful because they prevent faulty base objects from simulating client operations (read or write) that have not yet been invoked but will actually occur at some later point in time. Tokens are strictly weaker than signatures, because they cannot prevent a faulty base object from successfully forging a value that is never written. Consider for instance the lower bound of reading from a safe storage with optimal resilience [ACKM06]. It states that with t faulty objects, a read that does not modify the base objects takes at least $t + 1$ rounds before it can read a value. In each read round, a different malicious object simulates a concurrency with the same write, thereby triggering a new read round. With secret tokens, the second read round definitely reveals which value can be returned and the read terminates.

The assumption that tokens are secret can be violated with some probability. However, this probability can be arbitrarily reduced, for example, by uniformly and independently generating random tokens of k bits and by increasing the value of k. Note that in practice, assumptions generally hold only with a certain probability, e.g., the assumption that no more than t base objects fail.

Our first algorithm does not require readers to modify the base objects. As a consequence, it supports an unbounded number of possibly malicious readers. Every read completes after two communication rounds, which we show to be a tight bound. Thus, the algorithm improves on the read complexity of $t + 1$ rounds established for unauthenticated storage with optimal resilience when readers do not write [ACKM06]. Our second algorithm guarantees that every read is *fast*, by allowing readers to modify the base objects. The general lower bound of two rounds for reading from a robust storage with optimal resilience [GV06] is circumvented by having readers writing secret tokens into storage.

1.3.5 (C5) Robust Atomic Storage Complexity

As the final contribution of this thesis we determine the worst-case time complexity of robust atomic storage, which despite the wealth of research in distributed storage, is still an unsolved problem. We focus on *optimally resilient* robust (briefly robust) atomic storage and present two lower bounds on time complexity of reading from such a storage, answering question **Q5**. Together, our lower bounds imply that *there is no scalable* robust atomic storage implementation in which all reads complete in less than *four* rounds, where by scalable we mean constant time complexity.

The first lower bound, referred to as the *read lower bound*, demonstrates the impossibility of reading from robust MRSW atomic storage in two rounds. More precisely, we show that if the number of storage objects is at most $4t$ and if the number of readers R is greater than 3, then no MRSW atomic implementation may have all reads complete in two rounds.

Our proof scheme resembles that of [DGLC04] and relies on sequentially appending reads on a write operation, while progressively deleting the steps of a write and preceding read operations, exploiting asynchrony and possible failures. This deletion ultimately allows reusing readers and reaching an impossibility with as few as $R = 4$ readers. As none of these appended operations are concurrent under step contention, the impossibility also holds under the assumption of secret tokens, in which the adversary is unable to simulate step contention among operations.

Our second lower bound, referred to as the *write lower bound*, shows that if read operations are required to complete in three communication rounds, then the number of write rounds k is $\Omega(log(t))$. More precisely, we show that if the number of storage objects is at most $3t + \lfloor t/t_k \rfloor$, where $t_k \leqslant t$ and $R \geqslant k$, then no implementation of a MRSW atomic storage may have all reads complete in three rounds and all writes in $k \leqslant \lfloor \log(\lceil \frac{3t_k+1}{2} \rceil) \rfloor$ rounds. In a sense, our lower bound generalizes the write lower bound of [ACKM06], which proves our result for the special case of $k = 1$.

While using a similar approach as in showing the read lower bound, the write lower bound proof is much more involved and differs from our read lower bound proof in several key aspects. Due to the additional third read round, read steps cannot be entirely deleted, which prohibits the reuse of readers. Consequently, the number of supported readers R and the number of write rounds k are related ($R \geqslant k$). Furthermore, the proof relies on a set of malicious objects that forges critical steps of the write and of prior reads with respect to subsequent reads. This set grows with the number of

appended reads, relating the number of faulty objects t and the number of readers (which is at least k). At the heart of the proof we use a recurrent formula that relates t and k, similar to a Fibonacci sequence, which describes the exact relation between the two parameters. In its closed form, the formula transforms to the log function ($k = \Omega(log(t))$).

1.4 Roadmap

Chapter 2 of the thesis gives our system model and important definitions used in the remainder of the thesis. Chapter 3 presents the impossibility of one-step consensus with zero-degradation using Ω, two ways to circumvent the impossibility, and our latency-optimal atomic broadcast algorithm. Chapter 4 extends these results and presents our optimal and practical generalized consensus implementation. The algorithm combines optimal latency with optimal resilience and linear messages. In Chapter 5 we turn our attention to read/write storage and show that amnesic robust storage can be as fast as non-amnesic storage by ways of two algorithms. In Chapter 6 we introduce the notion of secret tokens to bridge the complexity of authenticated and unauthenticated storage. The resulting algorithms combine optimal latency with optimal resilience. Chapter 7 provides two lower bounds on the read complexity of robust atomic storage with optimal resilience. The thesis concludes in Chapter 8, which summarizes the contributions and opens some avenues for future research.

Chapter 2

Preliminaries

2.1 Model

In this section, we describe the asynchronous message-passing model assumed throughout the thesis except in Chapter 5, in which processes communicate through shared objects. Additional details necessary to describe the shared-memory-model used, are provided in Chapter 5.

We model processes as deterministic I/O Automata [LR89]. A distributed system consists of a set of processes and each pair of processes is interconnected with point-to-point communication channels and communicate via message-passing. The state of the communication channel between two processes p and q is viewed as a set of messages $mset$ containing messages that are sent but not yet received (p and q are called *ends* of the communication channel). We assume that every message has two tags which identify the sender and the receiver of the message.

A distributed algorithm A is modeled as a collection of deterministic automata, where A_p is the automaton assigned to process p. We say that a process p is *benign*, if p follows the automaton assigned to it. Note that in a crash-stop failure model all processes are benign. Computation proceeds in steps of A. A step of A is denoted by a pair of process id and message set $\langle p, M \rangle$ (M might be $\varnothing$). In step $sp = \langle p, M \rangle$, a benign process p atomically performs the following steps (we say that p takes step sp):

(*receive*) removes the messages in M from $mset$,

(*compute*) applies M and its current state st_p to A_p, which outputs a new state st'_p and a set of messages to be sent, and then p adopts st'_p as its new state,

(*send*) puts the output messages in $mset$.

We assume that the system is *asynchronous*: there is no bound on message propagation delays, nor on relative processing speeds. However, for ease of presentation we sometimes refer to a global clock not accessible by the processes.

We say that communication channels are *reliable* iff for every two benign processes p and q, if p sends a message m to q, and both p and q take an infinite number of steps, then q eventually receives m. More formally, if p puts m in $mset$ and q is the receiver of m, and both p and q take an infinite number of steps of their assigned automata A_p and A_q respectively, then there is a step $\langle q, M \rangle$ such that $m \in M$.

Given any algorithm A, a *run* (also called *execution*) of A is an infinite sequence of steps of A taken by benign processes, such that the following properties hold for each benign process p: (1) initially, for each benign process p, $mset = \varnothing$, (2) the current state in the first step of p is a special state $Init$, (3) for each step $\langle p, M \rangle$, and for every message $m \in M$, p is the receiver of m and $mset$ contains m immediately before the step $\langle p, M \rangle$ is taken.

A *partial run* is a finite prefix of some run. A (partial) run r extends some partial run pr if pr is a prefix of r. At the end of a partial run, all messages that are sent but not yet received are said to be *in transit*.

We say that a benign process p is *correct* (also called *non-faulty*) in a run r if p takes an infinite number of steps of A_p in r. Otherwise, a benign process p is *crash-faulty*. We say that a *crash-faulty* process p *crashes* at step sp in a run, if sp is the last step of p in that run.

In this thesis we distinguish two failure models, (1) the crash-stop failure model and (2) the Byzantine failure model [LSP82]. In the crash-stop model, every process is benign. A benign process is either correct or crash-faulty.

In the Byzantine failure model, a process is either benign or malicious (also called NR-Arbitrary [JCT98]). A *malicious* (or Byzantine) process p can perform arbitrary actions: p can remove or put messages in $mset$ at arbitrary times and can change its state in an arbitrary manner. However, p cannot put messages into (resp. remove messages from) a channel p is not an end of. In practice, this assumption is implemented using message authentication codes [Tsu92]. Malicious processes and benign processes which are crash-faulty are collectively called *faulty*.

2.2 Consensus

The distributed system we consider consists of a set of n processes of which up to f may fail by crashing. Precise assumptions on the number f of failed

processes are problem specific and are detailed in the respective Chapters. However, we say that an algorithm has optimal resilience if $n = 2f + 1$. Solving for f results in $\lfloor \frac{n-1}{2} \rfloor$ being the maximum number of faults any consensus algorithm can tolerate [CT96].

2.2.1 Traditional Consensus

In the traditional consensus problem, processes have to irrevocably agree on a value that is one of the values proposed by some process. Formally, traditional consensus is defined by two safety properties (Validity and Agreement) and one liveness property (Termination) [CT96]:

Validity: If a process decides v, then some process has proposed v.

Agreement: No two processes decide differently.

Termination: Every correct process decides.

This is the definition of consensus we use in Chapter 3. Given its simplicity, this very popular definition of consensus has been considered in many (mostly theoretical) works on consensus.

Asynchrony and crashes create a context in which consensus has no deterministic solution [FLP85]. As discussed in the introduction, a popular way to circumvent this impossibility is to add timing assumptions to the system model that are required to hold only eventually [DLS88].

2.2.2 Failure Detectors

Instead of dealing with low level details about synchrony and associated timing assumptions, failure detectors [CT96] are defined in terms of properties, allowing a clean separation from the implementation. We assume that the system is equipped with an appropriate distributed failure detector, consisting of one failure detector module installed at each process. The relevant failure detectors for this thesis are the *leader failure detector* Ω (also called *leader oracle*) and the *eventually perfect* failure detector $\diamond\mathcal{P}$. Both eventually provide consistent and correct information about the state of processes, i.e., crashed or non-crashed. While $\diamond\mathcal{P}$ eventually outputs exactly the crashed processes, Ω eventually outputs a single correct *leader* process. Ω is strictly weaker than $\diamond\mathcal{P}$ and it is the weakest failure detector to solve consensus [CHT96, Chu98]. More formally, $\diamond\mathcal{P}$ is defined in terms of the following two properties:

Eventual Strong Completeness: Eventually, every crashed process is suspected by every correct process.

Eventual Strong Accuracy: Eventually, no correct process is suspected by any correct process.

Ω is defined in terms of the eventual leadership property:

Eventual Leader: Eventually, Ω outputs the same correct process forever.

2.2.3 The Atomic Broadcast Problem

In the atomic broadcast problem processes have to agree on a unique sequence of messages. Formally, the atomic broadcast problem is defined in terms of two primitives a-broadcast(m) and a-deliver(m), where m is a message. When a process p executes a-broadcast(m) (respectively a-deliver(m)), we say that p a-broadcasts m (respectively p a-delivers m). We assume that every message m is uniquely identified and carries the identity of its sender. The atomic broadcast problem is defined by two liveness properties (Validity and Agreement) and two safety properties (Integrity and Total Order) [CT96]:

Validity: If a correct process a-broadcasts a message m, then it eventually a-delivers m.

Agreement: If a process a-delivers message m, then all correct processes eventually a-deliver m.

Integrity: For any message m, every process a-delivers m at most once, and only if m was previously a-broadcast.

Total Order: If some process a-delivers message m' after message m, then a process a-delivers m' only after it a-delivers m.

2.2.4 Spontaneous Total Order

As pointed out by Pedone and Schiper [PSUC02], messages broadcast in LANs are likely to be delivered totally ordered. This phenomenon can be attributed to the small variation of network delays in a LAN. Thus, if two distinct processes broadcast m and m' respectively, then it is very likely that m is delivered by all processes before m' or viceversa. The authors of [PSUC02] propose a new oracle called *Weak Atomic Broadcast* (WAB) that captures the property of spontaneous total order. A WAB oracle is

defined by the primitives w-broadcast(k,m) and w-deliver(k,m), where $k \in \mathbb{N}$ is the kth w-broadcast instance and m is a message. When a process p executes w-broadcast(k, m), we say that p w-broadcasts m in instance k. When a process p executes w-deliver(k, m) we say that p w-delivers m that was w-broadcast in instance k. Intuitively, if WAB is invoked infinitely often, it gives the same output to every process infinitely often. Formally, a WAB oracle satisfies the following properties:

Validity: If a correct process invokes w-broadcast(k, m), then all correct processes eventually output w-deliver(k, m).

Uniform Integrity: For every pair (k,m), w-deliver(k,m) is output at most once and only if some process invoked w-broadcast(k,m)

Spontaneous Order: If w-broadcast(j,$*$) is called for infinitely many different instances j then infinitely many instances k exist in which the first message w-delivered in instance k is the same for every process that w-delivers messages in k.

2.2.5 Revisiting Consensus in Lamport's Framework

In the state machine approach, a collection of servers executes a sequence of consensus instances to choose a sequence of client commands. A client sends a command to the servers, and the servers propose that command in the next instance of consensus. By considering only the cost of the consensus algorithm, the messages sent by the client are ignored. Lamport [Lam03] introduces a generalization of the traditional consensus framework, which accounts for all the costs (messages and delays) of state machine replication. Here, consensus is defined in terms of three types of agents:

Proposers: A proposer can propose values.

Acceptors: The acceptors cooperate to choose a single proposed value.

Learners: A learner can learn what value has been chosen.

The traditional consensus framework is somewhat rigid, in that these sets are equal and each process is proposer, acceptor and learner. Lamport's consensus framework provides more flexibility and allows to better model a client/server architecture in which each client can be considered to be both proposer and learner, and the servers to be acceptors.

We are now ready to restate the consensus problem as defined by Lamport [Lam06b]:

Nontriviality: Only a proposed value may be chosen.

Consistency: Any two values that are chosen must be equal.

Conservatism: If a learner learns value v, then v is chosen.

Progress: If p and l are correct and p proposes a value v, then l eventually learns v.

This definition can be applied to client/server systems in which clients (who can play the roles of proposer and learner) are not necessarily reliable. For instance, a client could invoke an operation and then vanish. Therefore, reliability assumptions are made only on acceptors. We reconsider a distributed system to consist of any number of proposers and learners and n acceptors of which at most f may crash.

2.2.6 Generalized Consensus

In an effort to define consensus in the way it is actually used in the state machine approach, Lamport [Lam05] extends the concept of consensus from agreement on a single value, to agreement on a dynamic set of values. This is done in two stages. In the first stage, consensus is expressed in terms of agreement on a growing *sequence* of commands. The observation that leads to the second stage is that ordering *commutable* commands is unnecessary. Instead of choosing a sequence of commands, it suffices to choose a partially ordered set of commands in which any two *interfering* (i.e. non-commuting) commands are ordered. Such a partially ordered set is called a *command history*. Executing the commands in a command history in any order consistent with its partial order has the same effect. Thus, a history defines an *equivalence class* of command sequences.

Histories are constructed by appending a command sequence σ to the initially empty history $\bot$ using the special append operator $\bullet$. The resulting history is $\bot \bullet \sigma$. Histories $\bot \bullet \sigma$ and $\bot \bullet \tau$ are equal iff σ and τ are equivalent command sequences.

The prefix relation $\sqsubseteq$ on the set of histories is defined as a partial order. For two histories h and h', $h \sqsubseteq h'$ iff there is a command sequence σ such that $h \bullet \sigma = h'$. We say that h is a prefix of h' (or equivalently that h' is an extension of h). A history h is isomorphic to a directed graph $G(h)$ whose nodes are the commands. There is an edge between any two interfering commands c_i and c_j from c_i to c_j in $G(h)$ iff $i < j$ in h. For two histories h and h', it holds that $h \sqsubseteq h'$ iff the graph $G(h)$ is a prefix of the graph $G(h')$. $G(h) = G(h')$ iff $h = h'$.

A lower bound of a set H of histories is a history that is a prefix of every element in H. The greatest lower bound (*glb*) of H is a lower bound of H that is an extension of every lower bound of H. We write the *glb* of H as $\bigsqcap H$ and we let $h \sqcap h'$ equal $\sqcap\{h, h'\}$ for any two histories h and h'. The least upper bound (*lub*) is defined in the analogous manner. We write *lub* of H as $\bigsqcup H$ and we let $h \sqcup h'$ equal $\sqcup\{h, h'\}$. Intuitively, the *glb* (resp. *lub*) of a set of histories is the largest common prefix (resp. the smallest common extension).

We define two histories h and h' to be *compatible* iff they have a common upper bound, i.e., there is some history g with $h \sqsubseteq g$ and $h' \sqsubseteq g$. A set of histories H is compatible iff every pair of histories in H are compatible.

We are now ready to state the properties of generalized consensus.

Nontriviality: If history h is chosen, then there exists a proposed command sequence σ, such that $h = \bot \bullet \sigma$

Consistency: If any two histories h and h' are chosen, then h and h' are compatible.

Conservatism: If a history h is learned, then h is chosen.

Progress: If p and l are correct and p proposes command c, then eventually l learns a history containing c.

2.2.7 Complexity Measures

As the main complexity measures characterizing the efficiency of consensus and atomic broadcast, we consider time and message complexity.

Time Complexity

Since we assume an asynchronous model, in the worst case, the latency of consensus (respectively atomic broadcast) is unbounded. Therefore, we measure latency in the *best case*. The best case is characterized by *stable* runs in which the failure detector used by the respective algorithm provides (a) accurate information about the correct/crashed processes and (b) its output does not change during the run.

In this thesis, we rely on the definition of *time complexity* for asynchronous algorithms from [Awe85, AW98] constrained to stable executions. We define the *propagation delay* of a message to be the time that elapses between the event that sends the message and the event that receives the message. The *time complexity* (or *latency*) of an algorithm is defined as the

maximum number of time units from the start until termination of the algorithm, taken over all stable executions, assuming that the *propagation delay* is one time unit. We refer to the latency of k time units as k *communication steps* (or equivalently k *message delays*).

The definition of termination is problem specific. In the traditional consensus (resp. atomic broadcast) problem we say that the algorithm terminates when all correct processes have decided (resp. have a-delivered the a-broadcast message). In generalized consensus, termination means that a correct learner l has learned a history containing a command c proposed by correct proposer p, where in our model, l and p are mapped to the same client process (see Chapter 4 for details).

Message Complexity We measure the message complexity of an algorithm as the maximum number of messages sent from the start until the termination of the algorithm, taken over all stable executions.

2.3 Distributed Storage

A distributed storage can be viewed as a read/write data structure implemented by two disjoint sets: (1) a set *objects* of n processes, called *base objects* (we sometimes refer to them as *servers*) and (2) a possibly unbounded set of processes called *clients*. Any number of clients may be faulty, whereas only a threshold t of base objects may fail. Precise statements on the relation between t and n are problem specific and are given in the respective Chapters. However, we say that an algorithm has *optimal resilience* when $n = 3t + 1$. Solving for t results in $\lfloor \frac{n-1}{3} \rfloor$, being the maximum number of base object failures a storage algorithm can tolerate [MAD02].

In the thesis we consider the fundamental class of multiple-reader single-writer (MRSW) storage, in which the set of clients consists of two disjoint subsets, a singleton *writer* and a possibly unbounded set *readers* with cardinality R (if bounded). In our model we assume that base objects are passive: (a) they send messages to clients only in reply to a request and (b) base objects do not communicate with each other. This model is in line with a large amount of recent work in distributed storage, motivated by the advent of storage area networks (SANs) and network attached storage (NAS), where base objects model active disks supporting read-modify-write operations [AW98].

In this thesis we assume the worst-case behavior of base objects, allowing base objects to fail Byzantine. However, we assume that clients fail by crashing. The reason for modeling clients as benign is that a Byzantine writer which is writing bogus values into the storage, and otherwise follows the pro-

tocol, may be undetectable. The same holds for a Byzantine reader that is allowed to write (back) values. Sometimes we can relax the assumption that readers are benign, for instance when dealing with regular storage, detailed in Chapters 5 and 6.

A read/write storage abstraction provides two operations: write(v), which stores v in the register, and read(), which returns the value from the register. We assume that each client invokes at most one operation at a time (i.e., it does not invoke the next operation until it receives the response for the current operation). Only readers invoke read operations and only the writer invokes write operations. We further assume that the initial value of a register is a special value $v_0 = \perp$, which is not a valid input value for a write operation. We say that an operation *op* is *complete* in a (partial) run if the run contains a response step for *op*. In any run, we say that a complete operation op_1 precedes operation op_2 (or op_2 succeeds op_1) if the response step of op_1 precedes the invocation step of op_2 in that run. If neither op_1 nor op_2 precedes the other, then the operations are said to be concurrent. We say of an operation which does not overlap with any write that it is *uncontended*.

2.3.1 Register Types

Lamport [Lam86] defines three types of a register, *safe*, *regular* and *atomic*, in increasing strength. A storage algorithm is safe, regular or atomic iff it satisfies the properties of *safety*, *regularity* and *atomicity* respectively. In the following we give definitions of safety, regularity and atomicity for single-writer registers. In the single-writer setting, the writes in a run have a natural ordering which corresponds to their physical order. Let wr_k denote the k^{th} write in a run ($k \geqslant 1$), and let v_k be the value written by the k^{th} write. Further, let $v_0 = \perp$.

We say that a partial run satisfies *safety* if every uncontended read operation returns the value written by the last preceding write. More formally, if rd is an uncontended read operation and rd returns v_k, then (a) there is a write operation w_k preceding rd or $v_k = v_0$ and (b) there is no $l > k$ such that w_l precedes rd (w_k is the last preceding write).

A (partial) run satisfies *regularity* if it satisfies safety and *every* read operation (contented or uncontended) returns the value of the last preceding write or a value written by one of the concurrent writes.

Finally, a (partial) run satisfies *atomicity* if it satisfies regularity and *no read inversion*. Roughly speaking, a read operation never returns an older value than the one returned by a preceding read operation. More formally, a partial run satisfies atomicity if the following properties hold: (1) if a read returns x then there is k such that $v_k = x$, (2) if a read rd is complete and it

succeeds some write wr_k ($k \geqslant 1$), then rd returns v_l such that $l \geqslant k$, (3) if a read rd returns v_k ($k \geqslant 1$), then wr_k either precedes rd or is concurrent with rd, and (4) if some read rd_1 returns v_k ($k \geqslant 0$) and a read rd_2 that succeeds rd_1 returns v_l, then $l \geqslant k$.

An algorithm implements a register if every run of the algorithm satisfies *wait-freedom* and the respective consistency property (i.e. *safety, regularity, atomicity*) of the register. Wait-freedom [Her91] states that if a process invokes an operation, then eventually, unless that process crashes, the operation completes (even if all other client processes have crashed).

Following the definition from [CGK07], we call a storage algorithm A *robust* if A wait-free implements a regular register from Byzantine base objects in the unauthenticated data model.

2.3.2 Time Complexity

In the context of distributed storage we focus on the *worst-case* time complexity of a register implementation, measured in terms of communication round-trips (or simply rounds). A round is defined as in [GNS09, LS02, EGM+09, DGLC04]:

Definition 1. *Client c performs a communication round* rnd *during operation* op *if the following conditions hold:*

1. *The client c sends messages to all objects. (Not sending messages to certain objects can be modeled by having these objects not change their state or reply).*

2. *Objects, on receiving such a message, reply to the reader (resp. the writer) before receiving any other messages.*

3. *When the invoking client receives replies from at most $n - t$ correct objects, the round (*rnd*) terminates (either completing the operation* op *or starting a new round).*

The *time complexity* (latency) of a distributed storage algorithm is defined as the maximum number of rounds taken over all possible executions. Note that a latency of k rounds is equivalent to $2k$ message delays.

Since up to t objects might be faulty, ideally, in every round *rnd* the invoking client c can only wait for reply messages from at most $n-t$ correct objects. If in a run r, a round *rnd* terminates based on replies from a set C of $n-t$ objects, then (a) either all objects in C are correct or (b) there is run r' indistinguishable to client c from r, in which all objects in C are correct.

Also, each round attempts to invoke operations on *all* objects. If on some correct object there is a pending invocation (of an earlier round), then the new invocation awaits the completion of the pending one. This notion of a round is equivalent to that in the model of [ACKM06].

Chapter 3

One-Step Consensus with Zero-Degradation

In this chapter we consider efficient implementations of consensus in the asynchronous model with crash-failures, enhanced with unreliable failure detectors. In such a setting, if all processes propose the same value, consensus is reached in one communication step. Assuming $f < n/3$, this is regardless of the failure detector output. A zero-degrading protocol reaches consensus in two communication steps in every stable run, i.e., when the failure detector makes no mistakes and its output does not change.

Our contribution is to show that leader based consensus implementations cannot be simultaneously one-step and zero-degrading. Also, we propose two approaches to circumvent the impossibility and present corresponding consensus protocols. Further, we describe a consensus-based atomic broadcast implementation which, using our consensus protocols, attains the optimal latency of three messages delays in every stable run and a latency of two in the absence of collisions. Collectively, our contributions provide answers to open research questions **Q1.1**, **Q1.2** and **Q1.3** raised in Section 1.2.1.

3.1 Introduction

As already motivated in Chapter 1, consensus is central to distributed system design, and many fault-tolerant coordination problems can be reduced to consensus. Specifically, atomic broadcast, which lies at the heart of state machine replication [Sch90] boils down to executing a sequence of consensus instances [CT96], one per message (or batch of messages).

If consensus was used only once (e.g. during initialization), then its performance wouldn't matter. However, consensus is used repeatedly, and thus

its *lantecy*, measured as the time elapsed until consensus is reached, is a critical performance indicator. Since the latency of consensus is unlimited in the worst case [FLP85], we focus on executions common in practice, with few failures and accurate failure detection.

Given that consensus is utilized in a repeated manner, the overhead caused by runs with failures is negligible. However, failures occurring during one instance of consensus can propagate as initial failures to *all* subsequent instances. Thus, we are interested in algorithms whose performance is not degraded in presence of initial failures. To characterize such algorithms, the notion of *stability* has been introduced. We say that a run is *stable* iff the failure detector makes no mistakes and its output does not change during that run. Algorithms that reach consensus with optimal latency (i.e. in two message delays) in every stable run are called *zero-degrading* [DG02].

Besides being latency optimal in the common case, we seek to expedite the decision when all processes propose the same value. Assuming $f < n/3$, no failure detector is therefore needed and *one* communication step is sufficient to achieve global decision.

3.1.1 Previous and Related Work

The idea of one-step consensus stems from Brasileiro et al. [BGMR01]. Although their solution is optimal when all proposals are equal, the protocol needs at least *three* communication steps when starting from other initial configurations. The algorithm goes through a preliminary voting phase in which processes exchange their proposals. If a process receives enough equal values it decides, otherwise it uses an underlying consensus module. If some process decides v after the first step, all processes that proceed without deciding propose v to the consensus module. Agreement is thus ensured by the properties of the underlying consensus. The drawback of this algorithm is that it needs three rounds from other initial configurations.

Based on Brasileiro's idea, Mostefaoui and Raynal [MR00] developed an atomic broadcast protocol that has two message delays in the best case but needs four in the normal case. Moreover, even if messages are ordered, it is very unlikely that all buffers have the same length when their content is proposed. Thus, distinct processes propose different values and the protocol works in the slower mode.

This problem was recognized by Pedone and Schiper [PS03] and they suggested agreement on the largest common prefix instead of agreement on the whole buffer. As long as all buffers share a nonempty common prefix of messages, their algorithm achieves a latency of two message delays. As soon as messages are out of order, consensus is needed, which adds a latency of

two additional message delays. This protocol tolerates a minority of faulty processes, but achieving a latency of 2δ requires collecting the proposals from *all* processes. Thus, even if a single process crashes, the protocol switches to the slower mode.

Based on the observation that in LANs, messages are frequently delivered in total order, Pedone and Schiper [PSUC02] introduced the notion of *ordering oracle* to model the spontaneous total order encountered in LANs. The authors present an atomic broadcast protocol that has a latency of two message delays in the absence of collisions, performing well in lightly loaded systems. However, their approach exhibits a dramatic performance degradation when the load is increased.

Recently, the authors of [CMP06] have extended the idea of weak ordering oracles to Paxos-like [Lam98] protocols. Paxos-like protocols allow for the recovery of crashed processes [ACT00] and are well suited for the client/server computation model. The R*-Consensus protocol of [CMP06] degrades if multiple clients issue requests concurrently and thus it suffers from the same drawback as the original [PSUC02].

The key assumption in Brasileiro's [BGMR01] one-step consensus is $f < n/3$. This is generalized by Lamport [Lam06b] who distinguishes between the number of correct processes required to reach consensus in one communication step ($n - e$ with $e \leqslant f$) and the number of correct processes needed for progress ($n - f$ with $f < n/2$). Intuitively, if a process p decides v in one communication step, then it has received $n - e$ equal values v. Consequently, every process q that receives a message from $n - f$ processes receives v $n - e - f$ times. Since among the $n - f$ values received by q at most e values are distinct from v, agreement is ensured if $n - e - f > e$. Thus, the degree of resilience is given by $n > max\{2f, 2e + f\}$. Maximizing e leads to $f < \lfloor n/3 \rfloor$, while maximizing f leads to $e \leqslant \lfloor n/4 \rfloor$.

Recently, Lamport has presented Fast Paxos [Lam06a], an improvement of the classic Paxos [Lam98] consensus protocol, that achieves a latency of two message delays in the absence of collisions. However, Fast Paxos has non-optimal latency if collisions are frequent. Also, if more than e processes have failed, Fast Paxos is slower than classic Paxos.

3.1.2 Contributions

The state of the art leaves the question open if there is a single consensus algorithm that is both one-step *and* zero-degrading. Thus, we ask the following: do one-step consensus protocols need *three* communication steps in general? In section 3.3 we show that no leader-based consensus protocol can be simultaneously one-step and zero-degrading. This implies that

leader-based algorithms reaching consensus in one communication step when all proposals are equal, require three communication steps in the common case [GR04].

Our subsequent goal is to find sufficient conditions for circumventing the established impossibility and to eliminate the incurred latency overhead. In this chapter we consider two different approaches and present corresponding consensus protocols. In the first approach, we condition one-step decision on the behavior of the failure detector. With this approach, one-step decision is guaranteed only in stable runs. The consensus algorithm we present in section 3.4 is both of practical and of theoretical interest. It is theoretically interesting because it uses the Ω failure detector, which is the weakest to solve consensus [CHT96]. Moreover, since stability frequently holds in practice, it is appealing to optimize the running time of consensus in stable executions. Our second approach is to assume a strictly stronger failure detector than Ω. The consensus protocol presented in Section 3.5 satisfies both one-step decision and zero-degradation and uses a $\diamond\mathcal{P}$ class failure detector.

Furthermore, in Section 3.6 we present a consensus based atomic broadcast algorithm that has a latency of 3δ in every stable run and a latency 2δ in case of no collisions, where δ is the maximum network delay. Finally, in Section 3.7 we present both analytical and experimental evaluations of our protocols.

3.2 Model

We now give an brief summary of the distributed system model formally defined in Chapter 2. We assume a crash-stop asynchronous distributed message-passing model consisting of a set of processes $\Pi = \{p_1, ..., p_n\}$ of which a subset of up to $f < \lfloor n/3 \rfloor$ may fail by crashing. A process that never crashes is *correct*, otherwise it is *faulty*. Processes communicate by sending and receiving messages over *reliable* channels. A reliable channel does not lose, duplicate or (undetectably) corrupt messages. Given two correct processes p and q, if p sends a message m to q then q eventually receives m. The system model is enhanced with failure detectors Ω and $\diamond P$. While Ω eventually outputs the same correct process to every correct process, $\diamond P$ eventually outputs exactly the faulty processes to every correct process.

3.3 The Lower Bound

In this section we prove a lower bound on consensus time complexity. We show that every one-step leader-based protocol has a run in which some process needs at least three communication steps to decide. In other words it is impossible to devise a leader-based consensus protocol that is one-step and zero-degrading. In order to develop an intuition for the impossibility result, we first describe Brasileiro's one-step consensus [BGMR01] and how we would have to combine it with a leader-based protocol to achieve zero-degradation.

In the first round of Brasileiro's one-step consensus, every process broadcasts its proposal and subsequently waits for a message from $n-f$ processes. A process p decides v iff it receives $n-f$ equal values v. Hence if a process p decides v, then every process q necessarily receives v at least $n-2f$ times. To ensure agreement, it is sufficient to require that v is a *majority* among the values received by q (which translates to $n-2f > f$).

If there are less than $n-f$ equal proposals, then the first round is wasted. To eliminate this overhead, one straightforward approach is to combine this scheme with the first round of a leader-based protocol. Here, consensus is reached in two communication steps if every correct process picks the leader value in the first round. Hence, in the combined protocol we have to ensure that if no process decides in the first round, then every correct process picks the leader value. However, this is only guaranteed if there are less than $n-2f$ equal proposals. Otherwise, it might happen that some process receives a majority value v and consequently picks v in order to ensure agreement, while some other process picks the leader value v_l and $v \neq v_l$. Hence, two distinct values are proposed in the second round and consequently some process might not decide before the third round.

Definition 2 (one-step)**.** *Assuming $f < n/3$, a consensus protocol is* one-step *iff it reaches consensus in one communication step in every run in which all proposals are equal.*

Definition 3 (stable run)**.** *A run of a consensus algorithm is* stable *iff the failure detector makes no mistakes and its output does not change during that run.*

The stability of the failure detector can be attributed to the fact that nearly all runs are synchronous and crashes are initial. Even if the failure detector needs to pass through a temporary stabilization period (e.g. after a failure), in most runs it will exhibit a stable and accurate behavior. In a stable run, Ω outputs the same correct process from the beginning of the run, while $\diamond\mathcal{P}$ suspects exactly the processes that have crashed initially.

Definition 4 (zero-degradation)**.** *A consensus algorithm $\mathcal{A}$ is* zero-degrading *iff $\mathcal{A}$ reaches consensus in two communication steps in every stable run.*

Theorem 1 (Lower Bound)**.** *Assuming $n \leqslant 4f$, every one-step consensus algorithm $\mathcal{A}$ based on Ω has a stable run in which some process decides after three communication steps or more.*

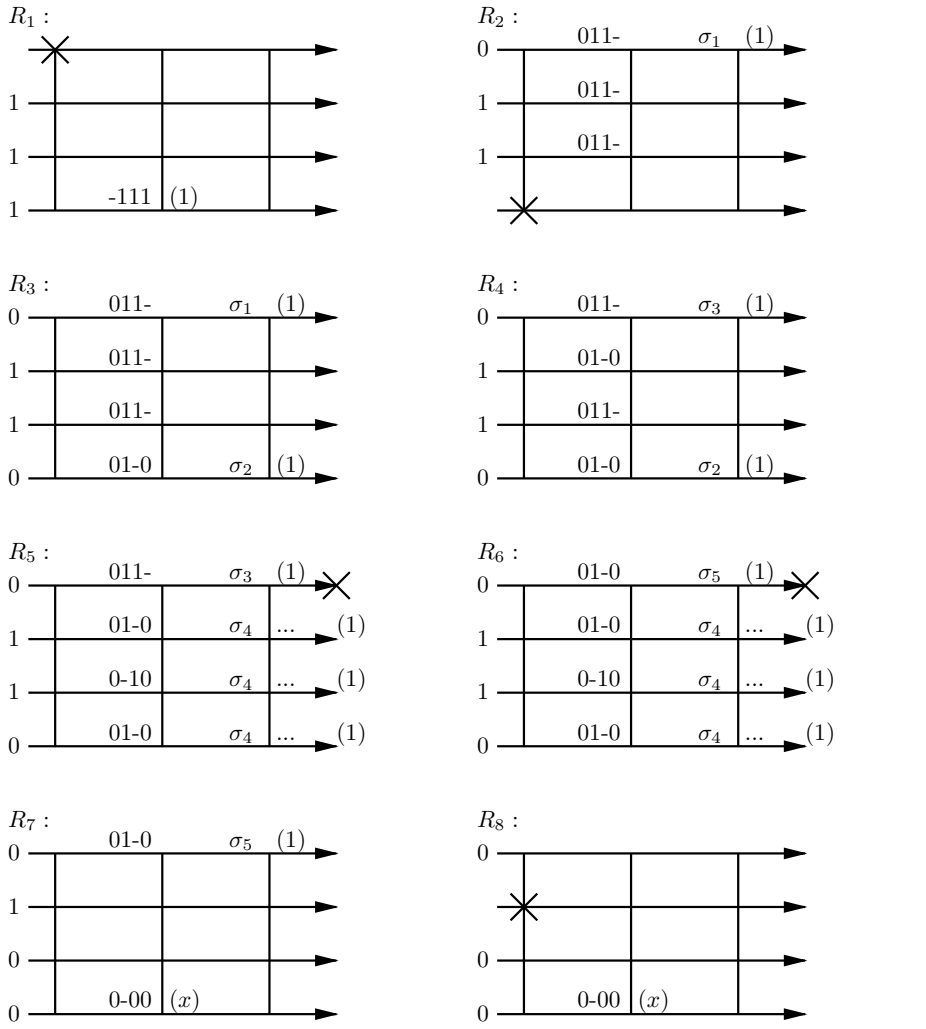
Proof. Preliminary notes (see Figure 3.1): We prove the theorem for the case $n = 4$ but this solution can be generalized to any value of n by employing the same technique as used in [GR04]. The state of a process after k communication steps is determined by its initial value, the failure detector output and the value and source of the messages received in every communication round up to k. To strengthen the result, the processes exchange their complete state. For the sake of simplicity, Ω outputs the same leader process p_1 at all processes in every run considered in the proof until p_1 possibly crashes. The state of process p after k communication steps denoted by is expressed as a k-dimensional vector with n entries such that the i-th entry contains the state of the i-th process after $k-1$ steps. Since in each round a process waits for a message from at most $n-f$ processes, one entry is empty. The decision value is bracketed $(0)/(1)$.

Two runs R_1 and R_2 are *similar* for process p up to step k, iff the state of p after k steps in R_1 is identical to the state of p after k steps in R_2. If two runs are similar for some process p, then p decides the same value in both runs.
Idea: The proof is by contradiction. We assume a leader-based one-step and zero-degrading protocol and show that it does not solve consensus. We construct a chain of possible runs such that every two neighboring runs are similar to some process. We start with a run in which all processes propose 1, and then we construct subsequent runs either by changing the communication pattern or the initial configuration. The failure detector assumption as well the expected properties of the protocol finally lead to the violation of one of the safety properties (validity or agreement).

- Since $\mathcal{A}$ is one-step, then it must have a run like R_1 in which all correct processes propose 1 and p_1 might have proposed the same. Thus, p_4 decides 1 after the first communication step[1].

- Since $\mathcal{A}$ is zero-degrading, then it must allow a run such as R_2. Run R_2 is stable because Ω outputs p_1 at all correct processes and its output does not change. Thus, p_1 decides after the second communication step.

[1] Actually, processes p_2 and p_3 also decide 1 after one communication step but this is not relevant for the proof.



$$\sigma_1 = \begin{bmatrix} 011\text{-} \\ 011\text{-} \\ 011\text{-} \\ - \end{bmatrix} \qquad \sigma_2 = \begin{bmatrix} 011\text{-} \\ - \\ 011\text{-} \\ 01\text{-}0 \end{bmatrix} \qquad \sigma_3 = \begin{bmatrix} 011\text{-} \\ 01\text{-}0 \\ - \\ 01\text{-}0 \end{bmatrix} \qquad \sigma_4 = \begin{bmatrix} - \\ 01\text{-}0 \\ 0\text{-}10 \\ 01\text{-}0 \end{bmatrix} \qquad \sigma_5 = \begin{bmatrix} 01\text{-}0 \\ 01\text{-}0 \\ - \\ 01\text{-}0 \end{bmatrix}$$

Figure 3.1: Illustration of the lower bound proof.

If p_1 decides 0, then we could construct a run R' that for p_1 is similar to R_2 (p_1 decides 0 in R') and that for p_4 is similar to R_1 (p_4 decides 1 in R'), violating agreement. Thus, in R_2, p_1 necessarily decides 1.

- Runs R_2 and R_3 are similar for p_1. Thus, p_1 decides 1 in R_3 after the second communication step. Since R_3 is stable, p_4 also decides 1 after the second step.

- Runs R_3 and R_4 are similar for p_4 and thus p_4 decides 1 in R_4 after the second communication step. Since R_4 is stable, p_1 also decides 1 after the second step.
- Runs R_4 and R_5 are similar for p_1. Consequently p_1 decides 1 in R_5 after the second communication step. In R_5 we crash p_1 so that all messages sent to p_2, p_3 and p_4 after the first communication step are lost. Since R_5 is not stable because Ω eventually outputs a new leader, p_2, p_3 and p_4 are only required to decide eventually. By agreement, they decide 1.
- In R_6 we crash p_1 such that R_5 and R_6 are similar for p_2, p_3 and p_4. Thus, they eventually decide 1. As p_1 cannot distinguish R_6 from a stable run, it decides after the second communication step. In order to ensure agreement, p_1 necessarily decides 1.
- Runs R_6 and R_7 are similar for p_1. Thus, p_1 decides 1 in R_7 after the second communication step.
- Since $\mathcal{A}$ is one-step, in run R_8, process p_4 decides x after the first communication step. Moreover, runs R_7 and R_8 are similar for p_4, and therefore p_4 also decides x in R_7.

There are two possible values for x. If $x = 0$ then agreement is violated in run R_7. Otherwise, if $x = 1$, then validity is violated in run R_8. □

3.4 Circumventing the Impossibility with Ω

In this section we present a leader-based consensus protocol that is zero-degrading but is not one-step, as this would contradict the established impossibility result. However, the protocol has the property that it reaches consensus in one communication step if all proposals are equal *and* the run is stable. The main idea behind the proposed **L**-Consensus algorithm depicted in Figure 3.2 is to constrain the processes to decide the value proposed by the leader. A process decides v in the first round if $n - f$ values including the leader value are equal to v. Consequently, every process that does not decide can safely pick the leader value. Hence, consensus is achieved in two rounds in every stable run. If there is no leader, then safety is ensured by picking the majority value.

The protocol executes in a round by round fashion. In every round, processes exchange messages, update their state depending on the messages received and possibly decide or move to the next round. The algorithm

```
Function Consensus(v_i)
  start T1, T2;
  task T1: r_i ← 1; est_i ← v_i; ld ← ⊥;
    while true do
      ld ← Ω.leader;
1      ∀j do send PROP(r_i, est_i, ld) to p_j enddo;
2      wait until received PROP(r_i, *, *) from n − f processes;
3      wait until received PROP(r_i, *, *) from ld ∨ ld ≠ Ω.leader;
4      if received PROP(r_i, v, ld) from n − f processes ∧ received
       PROP(r_i, v, *) from ld then
5          ∀j do send DECIDE(v) to p_j enddo;
6          return v;
7      else if received PROP(r_i, *, ld) from > n/2 processes ∧
       received PROP(r_i, v, *) from ld then
8          est_i ← v;
9      else if received PROP(r_i, v, *) from n − 2f processes then
10         est_i ← v;
       r_i ← r_i + 1;
11 task T2: upon reception of DECIDE(v): ∀j ≠ i do send
   DECIDE(v) to p_j enddo; return v;
```

Figure 3.2: The **L**-Consensus Algorithm

has three blocks that a process can execute in a round depending on which condition is satisfied (at line 4, 7 or 9). Safety is ensured as follows: if a process p decides a value v during round k, every process q that finishes round k, does so with value v, no matter what block it executes. In a stable run, the condition at line 7 evaluates to true, every correct process accepts the leader value and hence decides in the next round. In asynchronous runs, when there might be multiple leaders in the system, agreement is kept through majority voting. Since $n - f$ equal values are necessary for a decision, if a process decides v then every process receives v at least $n - 2f$ times, making the condition at line 9 become true. Since $n - 2f > f$, a process can safely pick the majority value.

3.4.1 Detailed Description

The **L**-Consensus algorithm consists of two parallel tasks $T1$ and $T2$. When a process p_i calls the `Consensus` function with a proposal v_i (i.e. it proposes value v_i), it initiates both tasks. Compliant with the definition of consensus, the `Consensus` function eventually returns the same decision value v to each non-crashed process.

Task 1: The algorithm executes a sequence of asynchronous rounds of one communication step each. In each round k, a process sends a round k message containing its current proposal to all processes and waits for round k messages from $n-f$ processes including its current leader, computes its new state based on the messages received (possibly decides), and moves to the next round. A process p_i maintains three local variables: the round number r_i, an estimate of the decision value est_i initialized to the proposal value v_i, and the current leader ld, initially $\perp$.

At the beginning of each round, p_i queries Ω for the current leader and stores the identity in ld. We say that p_i has leader p_l in round k if p_i sends a message with $ld = l$. The messages sent contain the following fields: k_i, est_i, ld. We say that a process p_l is *majority leader* for round k if a majority of processes send round k messages with $ld = l$. As any two majorities have a non empty intersection, there can be at most one majority leader at round k. Note that in asynchronous runs there are periods with no majority leader.

A process p_i can send two different types of messages in round k. If p_i has decided, then it broadcasts a decision value, otherwise it broadcasts a PROP(k_i, est_i, ld) message and we say that p_i proposes est_i in round k_i.

At the end of round k (i.e. after receiving round k messages from $n-f$ processes possibly including one from ld), process p_i updates its est_i variable as follows: if p_i receives a value v from the majority leader of round k, then $est_i = v$. If there is no majority leader or the Ω module at p_i suspects ld for having crashed and p_i receives $n-2f$ equal values v, then p_i picks v. Otherwise the estimate value is kept unchanged. A process p_i decides in round k if it receives $n-f$ equal values including one value from the majority leader.

Task 2: Upon receiving a decision message with value v, p_i forwards the decision value to the other processes and then decides v. Thus, if a correct process decides, the remaining correct processes cannot block since they eventually receive the decision message.

3.4.2 Correctness

Lemma 1 (Termination)**.** *Every correct process decides.*

Proof. We show that if some correct process never decides then every correct process eventually decides; a contradiction. If some correct process never decides then either some correct process decides or no correct process decides.
1) Case *a*: Some correct process decides. Then, it broadcast a decision message (line 5). Since it is correct, every correct process eventually receives the decision message (line 11) and also decides. Thus, every correct process decides, which contradicts the assumption.
2) Case *b*: No correct process decides. If some correct process p_i never decides, then either it is blocked in a round or it executes an infinite number of rounds.

Case 1: p_i blocks forever in a round. Let k be the first round in which some correct process is blocked. p_i can only be blocked at one of the wait statements (line 2 or 3).
- Case I: p_i is blocked at line 2 of round k. Since k is the first round in which some correct process blocks at line 2, all correct processes have broadcast a round k message at line 1. As communication links are reliable and there are at least $n-f$ correct processes, p_i eventually receives $n-f$ round k messages and completes line 2.
- Case II: p_i is blocked at line 3 of round k. As in the case above, every correct process broadcasts a round k message. Consider ld, which is the leader process output by Ω at p_i. If ld is correct, then p_i eventually receives a round k message from ld and completes line 3. Otherwise, if ld is faulty, then either p_i eventually receives a round k message from ld, or Ω eventually outputs a correct process different from ld and p_i completes line 3. Thus, p_i cannot block at line 3.

Case 2: All correct processes execute an infinite number of rounds without deciding. From the definition of a faulty process, there is a time t_1 such that every faulty process has crashed before t_1. From the definition of Ω there is a time t_2 such that Ω outputs the same correct process p_l at every correct process forever. Let $t := max\{t_1, t_2\}$ and k be the first round after t. In round k, every correct process sets ld to l and sends a message $(k, *, l)$ to all processes. Since no correct process decides, no correct process executes line 5. As there is a majority of correct processes and p_l is not suspected by any correct process, every correct process receives a majority of round k messages including one message from p_l, and every correct process sets its *est* variable to the same value (line 8). Therefore, at round $k+1$ every process including p_l sends a $(k+1, v, l)$ message. Thus, at round $k+1$ every correct process receives $n-f$ equal messages including a $(k+1, v, l)$ message from p_l. Therefore, the condition at line 4 evaluates to true and every correct process decides at line 5; a contradiction. □

Lemma 2 (Agreement)**.** *No two processes decide differently.*

Proof. A process can decide either at line 5 of some round or at line 11 of task $T2$. If a process decides v at line 11, then some other process has decided v at line 5. Let k be the lowest round in which some process p decides v at line 5. We claim that each process that decides v at line 5 of round k decides v, and that every process that completes round k does so with $est = v$. This implies that the est value of every process after round k is always v. Thus, in round k and after round k, v is the only value that can be decided at line 5. As k is the lowest round in which some process decides, this implies that v is the only value that can be decided in a round at line 5. This also implies that no process decides a value different from v at line 11 of task $T2$. Now we prove the above claim. Suppose that a process $q \neq p$ decides d in round k. Since $n - f > n/2$, both p and q receive equal values v and d respectively from a majority of processes. As any two majorities intersect in at least one process, it follows that $d = v$. Now, consider any process q' that completes round k without deciding. We show that q' completes round k with $est = v$. There are two cases to consider:

Case 1: q' evaluates the condition at line 7 to false. We show that q necessarily evaluates the condition at line 9 to true. At round k there are at least $n-f$ values v and q' has received $n-f$ values at line 2 of round k. Any two sets of $n-f$ elements have $n-2f$ elements in common, thus among the $n-f$ values q' receives at round k, at least $n-2f$ values are equal to v and at most f values are distinct from v. Since $n-2f > f$, v is a majority value among the values received by q'. Value v is unique as there cannot be two distinct majority values. Thus q' completes round k with $est = v$.

Case 2: q' evaluates the condition at line 7 to true. Thus, there must be a process p_l such that a majority of processes send messages with $ld = l$. Since p decides in round k, there must be a process $p_{l'}$, such that $n - f$ processes send messages with $ld = l'$. As any two majorities have a process in common, it follows that $l = l'$. Thus q completes round k with $est = v$. □

3.5 Circumventing the Impossibility with $\diamond\mathcal{P}$

In this section we present a one-step and zero-degrading algorithm that uses the $\diamond\mathcal{P}$ failure detector. The proposed **P**-Consensus algorithm illustrated in Figure 3.3 is based on a simple observation that was originally made by Lamport [Lam06a]. One of the necessary conditions for the impossibility of section 3.3 is that processes receive messages from different quorums in the first communication round. If all processes received the same set of messages,

then they could deterministically pick the same value to propose in the second round. Consequently, consensus would be obtained in two steps.

The idea behind **P**-Consensus is to use the $\diamond\mathcal{P}$ failure detector to build a consistent quorum from which every process delivers first round messages in case it cannot decide. In every stable run, $\diamond\mathcal{P}$ suspects exactly the faulty processes and its output does not change during that run. Hence, every process that does not decide during the first round computes the same quorum (line 5) and subsequently receives a message from every quorum member. The sets of messages received by different processes from the quorum are equal and the functions applied to pick a value are deterministic (lines 9-12). Hence, all processes start the next round with the same value and consequently every correct process decides in the second round.

3.5.1 Detailed Description

The **P**-Consensus algorithm consists of two parallel tasks $T1$ and $T2$ that are initiated when a process proposes a value. The `Consensus` function eventually returns the same decision value to every correct process. Since the second task is identical to task $T2$ of the **L**-Consensus protocol, we confine ourselves to describing task $T1$.

The algorithm executes a sequence of asynchronous rounds of one communication step each. In each round k, a process sends a round k message containing its current proposal to all processes and waits for round k messages from $n - f$ distinct processes, computes its new state based on the messages received and tries to decide. If it cannot decide then it possibly waits for more messages, computes its new state and moves to the next round.

A process p_i maintains two local variables: the round number k_i initialized to 1 and an estimate of the decision value est_i initialized to the proposal value v_i. At the beginning of each round, p_i broadcasts a message that contains the following fields: k_i, est_i. A process p_i can send two different types of messages in round k_i. If p_i has decided, then it broadcasts a decision value, otherwise it sends a PROP(k_i, est_i) message to all processes and we say that p_i proposes est_i in round k_i.

Subsequently, p_i waits for a message from $n - f$ distinct processes. If p_i receives $n - f$ identical values it decides. Otherwise, p_i additionally waits for messages from a quorum Q that is computed deterministically as the set that contains the first $n - f$ nonsuspected processes. We say that Q is *complete* iff it has $n - f$ members.

At the end of round k, p_i updates its est_i variable as follows: if there is a complete quorum Q such that p_i receives a message from each process in Q and there is a majority value v among the $n - f$ values received, then

```
Function Consensus(v_i)
  start T1, T2;
  task T1: r_i ← 0; est_i ← v_i;
    while true do
1       ∀j do send PROP(r_i, est_i) to p_j enddo;
2       wait until received PROP(r_i, *) from n − f processes;
3       if received PROP(r_i, v) from n − f processes then
4           ∀j do send DECIDE(v) to p_j enddo; return v;
5       let Q_i = { the first n − f processes p_j : j ∉ ◇P.suspected };
6       wait until received PROP(r_i, *) from every
        p_j : j ∈ Q_i \ ◇P.suspected;
7       let Qlist_i = (v | PROP(r_i, v) has been received from
        p_j : j ∈ Q_i);
8       if |Qlist_i| = n − f then
9           if ∃v ∈ Qlist_i : #(v) ⩾ n − 2f then
10              est_i ← v;
11          else
12              est_i ← est_{min{j|j∈Q_i}};
        else %ensure agreement%
13          let vlist_i = (v | PROP(r_i, v) has been received);
14          if ∃v ∈ vlist_i : #(v) > |vlist_i|/2 then
15              est_i ← v;
        r_i ← r_i + 1;
16  task T2: upon reception of DECIDE(v): ∀j ≠ i do send
    DECIDE(v) to p_j enddo; return v;
```

Figure 3.3: The **P**-Consensus Algorithm

$est_i = v$. If there is no such value v, then no process decided in round k. Thus, p can propose any value in the next round. Subsequently, p picks the estimate of the *leader*, the process with the smallest index among all nonsuspected processes. In case that there is no such process, p simply keeps its estimate. If Q is not complete and there is a majority value v among the values received in round k then $est_i = v$. If no such value exists, then p_i moves to the next round.

3.5.2 Correctness

Lemma 3 (Termination)**.** *Every correct process decides.*

Proof. We follow the same strategy as in Section 3.4.2 and show that if some correct process never decides then every correct process eventually decides. Assuming that some correct process never decides yields two cases. Either some correct process decides or no correct process decides. The latter case implies that some correct process never decides. Thus, either 1) it is blocked in a round or 2) it executes an infinite number of rounds.

- Case 1): The proof is similar to the one of Section 3.4.2. A process cannot block at one of the wait statements (at lines 2, 6) because at most f processes are faulty and $\diamond\mathcal{P}.suspected$ eventually contains all crashed processes.

- Case 2): All correct processes execute an infinite number of rounds without deciding. From the definition of a faulty process, there is a time t_1 such that every faulty process has crashed before t_1. From the definition of $\diamond\mathcal{P}$ there is a time t_2 such that after t_2, $\diamond\mathcal{P}$ outputs exactly the crashed processes forever. Let $t := max\{t_1, t_2\}$ and k be the first round after t. Since no correct process decides, no correct process executes line 4 and every correct process executes lines 5, 6 and 7. As $\diamond\mathcal{P}$ behaves perfectly in round k, every quorum Q contains exactly the correct processes. The fact that Q is complete and identical and every correct process receives a message from every member of Q implies that $Qlist$ is the same at every correct process and that $|Qlist| = n - f$. Hence, the condition at line 8 evaluates to true and all correct processes pick the same value either at line 10 or at line 12. Therefore, in round $k+1$, all correct processes send a message with the same value and hence every correct process receives $n - f$ identical values and consequently decides at line 4; a contradiction. □

Lemma 4 (Agreement)**.** *No two processes decide differently.*

Proof. We claim that each process that decides at line 4 of round k decides v, and that every process that completes round k without deciding does so with $est = v$. As already shown in 3.4.2, if this claim is true then agreement holds. Now, we prove the above claim. It is easy to see that if two distinct processes p and q decide in round k, then they decide the same value v. Let q' be a correct process that does not decide in round k. As q' receives at least $x \geqslant n - f$ messages, it receives at most f values $w \neq v$. Since $x - f \geqslant n - 2f > f$, v is a majority among the values received by q' in round k which implies that one of the conditions at line 9 or 14 evaluates to true. Thus, q' completes round k with $est = v$, which concludes the proof. □

3.6 The Atomic Broadcast Protocol

The proposed **C**-Abcast protocol in Figure 3.4 represents a modification of the WABcast atomic broadcast algorithm of [PSUC02]. Like the Chandra & Toueg's (CT) Atomic Broadcast protocol [CT96], **C**-Abcast reduces atomic broadcast to consensus. It executes a series of consensus instances to determine a single message delivery sequence at all processes. Unlike the CT Atomic Broadcast, **C**-Abcast assumes an underlying consensus module that is very efficient in case that all proposals are equal. In order to exploit the efficiency of the underlying consensus, **C**-Abcast uses a WAB oracle to provide the consensus module with equal input values. When the oracle outputs the same proposal to every process, **C**-Abcast has a latency of two message delays, i.e., 2δ; one for asking the oracle plus one communication step for consensus. In case of collisions, consensus is reached in two communication steps. Hence, **C**-Abcast has a latency of three message delays, i.e, 3δ in the common case.

The protocol consists of three concurrent tasks. A process can either a-broadcast a message (line 2), a-deliver a message (line 4), or w-deliver a message (line 16). A process p a-broadcasts a message m by including m in a set $estimate_p$. This set contains the messages that have not been yet a-delivered by p. The a-deliver($*$) task executes in a round by round fashion. In round k, process p w-broadcasts the set $estimate_p$ and waits to w-deliver the first value v output by its oracle. Then, p proposes v to the k-th consensus instance and waits for the decision. After it decides, p atomically delivers all messages contained in the k-th decision in some deterministic order, removes from $estimate_p$ every message a-delivered so far and moves to the next round. In order to ensure validity, every message a-broadcast by some correct process must eventually be contained in the proposal of every correct process. Thus, in the third task (line 16), every process p includes in $estimate_p$ all messages w-broadcast so far.

3.6.1 Correctness

Informally, Lemma 5 below states that $\forall k \in \mathbb{N}$, (a) if a process delivers the k-th message batch, then every correct process also delivers it and (b) that the k-th message batch is the same at every process. From (a) and (b) we can easily deduce Agreement and Total Order. Validity requires a more detailed proof.

Lemma 5. *For all $k > 0$, every process p and every correct process q, if p executes round k until the end then q executes round k until the end and $adeliver_p^k = adeliver_q^k$.*

```
   Initialization:
1    k_i ← 1; estimate_i ← ⊥; adelivered_i ← ⊥;
2  a-broadcast(m):
3    estimate_i ← estimate_i ∪ {m};
4  a-deliver(∗):
5    while true do
6        w-broadcast(k_i, estimate_i);
7        wait until w-deliver of the first message (k_i, v);
8        msgSet_i ← Consensus(k_i, v);
9      adeliver_i ← msgSet_i − adelivered_i;
10     deliver all messages in adeliver_i atomically in some deterministic
       order;
11     adelivered_i ← adelivered_i ∪ adeliver_i;
12      estimate_i ← estimate_i − adelivered_i;
13     k_i ← k_i + 1;
14     if estimate_i = ∅ then
15          wait until w-deliver of the first message (k_i, v)
           ∨ estimate_i ≠ ∅
16 upon w-deliver(∗, v) of the second, third etc. message of any round
17   estimate_i ← estimate_i ∪ v;
```

Figure 3.4: The **C**-Abcast Algorithm

Proof. We will prove the lemma by induction over k. First, it is easy to see that every correct process executes round 1 until the end. Due to consensus agreement, if p a-delivers messages in round 1 then $adeliver_p^1 = adeliver_q^1$. Now assume that the lemma holds for all k, $1 \leqslant k < r$. We first show that if p a-delivers messages in round r then q executes round r until the end. If p a-delivers messages in round r, then p returns from the invocation of Consensus$(r, *)$ at line 8. Since there is at most a minority of faulty processes, at least one correct process u executes Consensus$(r, *)$. This implies that u w-broadcasts its estimate at line 6. By the induction hypothesis, if p a-delivers messages in round $r-1$, q executes round $r-1$ until the end. Thus, q eventually w-delivers the first message of stage r either a) at line 7 or b) at line 15. Without loss of generality, let $estimate_u$ be the first message w-delivered by q in round r. In both cases q breaks from the corresponding wait statement and executes Consensus$(r, estimate_u)$[2]. By consensus

[2]In case q breaks from the second wait statement (line 15) it does not block at the first

termination, q eventually executes round r until the end.

We now show that if p a-delivers messages in round r then $adeliver_p^r = adeliver_q^r$. As shown in the first part of the lemma, q executes round r until the end. Thus, q a-delivers messages in $adeliver_q^r$. Due to consensus agreement $msgSet_p^r = msgSet_q^r$. By the induction hypothesis, $\forall k, 1 \leqslant k < r : adeliver_p^k = adeliver_q^k \Rightarrow \cup_{k=1}^{r-1} adeliver_p^k = \cup_{k=1}^{r-1} adeliver_q^k$. As $adeliver^r = msgSet^r - \cup_{k=1}^{r-1} adeliver^k$, we get $adeliver_p^r = msgSet_p^r - \cup_{k=1}^{r-1} adeliver_p^k = msgSet_q^r - \cup_{k=1}^{r-1} adeliver_q^k = adeliver_q^r$. □

Lemma 6 (Agreement)**.** *If a process a-delivers message m, then all correct processes eventually a-deliver m.*

Proof. Follows directly from Lemma 5. □

Lemma 7 (Total Order)**.** *If some process a-delivers message m' after message m, then a process a-delivers m' only after it a-delivers m.*

Proof. Follows from lemma 5, the total ordering of natural numbers, and the fact that messages within a batch are delivered atomically in a deterministic order. □

Lemma 8 (Validity)**.** *If a correct process a-broadcasts message m, then eventually it a-delivers m.*

Proof. The proof is by contradiction. Suppose that a correct process a-broadcasts m but never a-delivers m. By Lemma 6 no correct process a-delivers m. Consider a process p that a-broadcasts a message m. Consequently, p includes m in $estimate_p$ and thus w-broadcasts m. By the validity property of the ordering oracle, every correct process eventually w-delivers m at line 16 and thus includes m in its *estimate*. Since no correct process adelivers m, no correct process removes m from its *estimate* at line 12. There is a time t so that all faulty processes have crashed before t and at which m is included in the *estimate* of every correct process. Let k be the lowest round number after t. Every correct process w-broadcasts m in k, which implies that every value proposed to the k-th consensus instance necessarily contains m. Due to validity of consensus, m is included in the $msgSet$ of every correct process. Thus, m is a-delivered by every correct process at round k; a contradiction. □

---

wait statement (line 7) because it has already w-delivered the first round r message.

3.7 Performance Evaluation

In this section we provide a brief comparison of our protocols with Paxos and WABcast, both analytical and experimental. Table 3.1 matches our protocols with Paxos [Lam98] and WABCast [PSUC02] in terms of latency (where δ is the maximum network delay), message complexity, resilience, and the oracle or failure detector used. In the absence of collisions, WABCast and **C**-

Table 3.1: Comparison of various atomic broadcast protocols

Protocol	No Collisions ; Collisions		Resil.	Oracle
	latency	#messages		
Paxos	3δ	n^2+n+1	$f < n/2$	Ω
WABCast	2δ ; ∞	n^2+n ; ∞	$f < n/3$	WAB
C-Abcast (**L**-/**P**-Cons.)	2δ ; 3δ	n^2+n ; $2n^2+n$		$\Omega/\diamond\mathcal{P}$

Abcast have the same time and message complexity. However, compared to Paxos, **C**-Abcast trades optimal resilience for improved latency. In presence of collisions, WABCast might not terminate whereas **C**-Abcast has the same time complexity as Paxos, albeit with increased messages.

3.7.1 Experimental Evaluation

We now present the experimental comparison of our protocols with Paxos and WABCast. We measure the latency of atomic broadcast as a function of the throughput, whereby latency is defined as the shortest delay between a-broadcasting a message m and a-delivering m. We implemented **L**-/**P**-Consensus and **C**-Abcast using the Neko [UDS01] framework. The experiments were conducted on a cluster of 4 identical workstations (2.8GHz, 512MB) interconnected by a 100Mb ethernet LAN. The different consensus algorithms were evaluated by exchanging the consensus module of **C**-Abcast. The WAB oracle implementation uses UDP packets whereas the rest of the communication is TCP-based. We considered only stable runs in our experiments. In order to capture the performance with and without collisions, we varied the load between $20msg/s$ and $500msg/s$. Figure 3.5 shows the comparison of our protocols with WABCast. Both proposed protocols exhibit a comparable latency to WABCast up to a load of $80msg/s$ and they dramatically outperform WABCast for loads higher than $100msg/s$.

Figure 3.6 summarizes the comparison with Paxos. In the lower half of the load spectrum, where collisions are rare, our protocols slightly outperform Paxos. However, starting from a load of $300msg/s$ upwards, where collisions

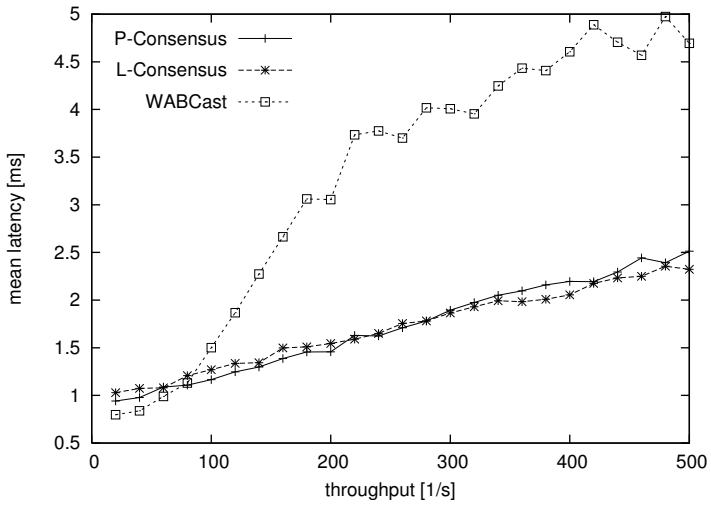


Figure 3.5: **C**-Abcast using **L**-/**P**-Cons. vs. WABcast ($n = 4$)

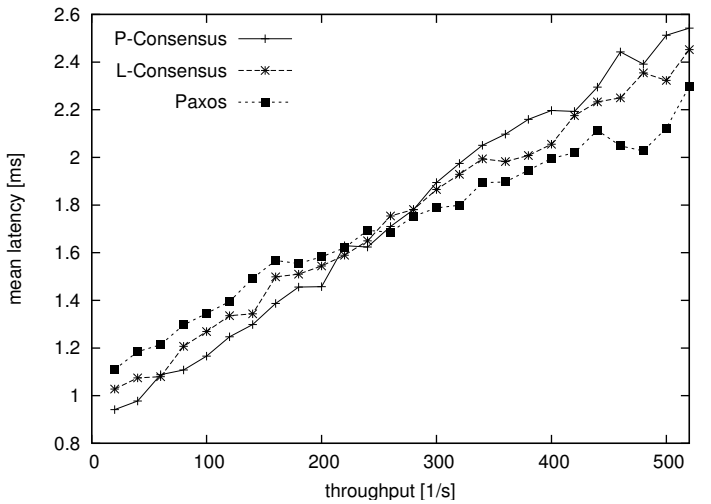


Figure 3.6: **C**-Abcast using **L**-/**P**-Cons. ($n = 4$) vs. Paxos ($n = 3$)

start predominating, Paxos performs better. This can be attributed to the high message overhead incurred **L**-/**P**-Consensus.

3.8 Summary of the Results

One-step decision and zero-degradation are desirable efficiency properties of consensus, and protocols satisfying them are latency optimal. We have investigated the compatibility of these two properties and have shown that

they cannot be both satisfied using the Ω failure detector. Subsequently, we have proposed two different approaches to circumvent the established impossibility. The first approach relaxes one-step decision to hold only in stable runs, while the second one assumes a strictly stronger failure detector. For each of the two approaches we have given a corresponding algorithm. While the proposed **L**-Consensus guarantees one-step decision only in stable runs, **P**-Consensus decides after one communication step regardless of the failure detector output. When compared with Paxos, our protocols exhibit improved latency for the lower half of the load spectrum. However, when the load increases and collisions prevail, the high message complexity incurred by our protocols leads to a noticeable performance degradation.

Chapter 4

Generalized Consensus and Hybrid Paxos

This chapter extends our previous results on latency-optimal consensus implementations presented in Chapter 3, to generalized consensus, where collisions are generated only by conflicting commands. Also it investigates the applicability of generalized consensus to wide area networks (WANs).

Our contribution is a generalized consensus protocol called Hybrid Paxos (HP) that matches all known lower bounds on latency, resilience and messages. In the absence of collisions caused by interfering commands, HP requires two message delays, and only one extra delay in the common case, which is optimal. Our experimental comparison with Classic Paxos (CP) confirms that HP (a) features a dramatic improvement in latency, (b) offers better latency up to 70% of CP's peak throughput, and (c) never performs worse. Altogether, our results answer question **Q2** in the affirmative.

4.1 Introduction

As identified in Chapter 3, frequent collisions can cause a significant performance degradation in practice, even if the algorithm employed is latency optimal in theory. Unlike LANs, in a generic environment such as a WAN, asymmetric and oscillating link delays combined with concurrent client invocations, result in collisions predominating even in lightly loaded systems.

WAN replication is appealing because it offers protection against catastrophic failures of a single site and can be used to enhance the resilience of critical services. To provide sustained performance in a WAN environment, it is imperative to minimize the occurrence of collisions. In this chapter, we turn to generalized consensus [Lam05], that distinguishes between *com-*

mutable and *non-commutable* (also called *interfering*) commands. The advantage of generalized consensus is that only interfering commands generate collisions. Thus, if many commands commute, by ordering only the fraction of interfering commands, the actual number of collisions can be reduced.

In the standard state-machine approach, a sequence of instances of a traditional consensus protocol are used to choose the sequence of client commands, where the ith instance chooses the ith command. In generalized consensus [Lam05], a *single* instance of consensus is used to choose an increasing *history* of commands. A history is a partially ordered set of commands, in which every pair of interfering commands is ordered.

As discussed earlier in Section 2.2.5, the generalized consensus problem is stated in terms of *proposers* that propose commands, *acceptors* that choose an increasing command history and *learners* that learn what history has been chosen. In a client/server system, clients might play the roles of proposer and learner and servers might play the role(s) of acceptor (and learner). A leader is elected among the acceptors to coordinate their actions.

Consensus protocols attaining the optimal latency [Lam06b] are the well known Classic Paxos (CP) [Lam98] and the more recent Generalized Paxos (GP) [Lam05]. Their message patterns are illustrated in Figure 4.1. In normal operation, CP requires three message delays. The communication pattern during normal operation is Client → Leader → Acceptors → Learners. GP saves one message delay by having the clients send their proposals directly to the acceptors, bypassing the leader. This works fine if the acceptors receive the same sequence of interfering commands. However, when commands are proposed concurrently, interfering commands may be accepted in different orders, resulting in no command being chosen. In order to guarantee progress, GP then runs a collision recovery procedure, which adds *four* message delays. Thus, if collisions are frequent, GP has a significantly higher latency and a lower throughput than CP.

We found that even in the absence of collisions, depending on the layout of clients and servers, CP can outperform GP (for many clients). This stems from the fact that in order to be fast, GP needs larger quorums than CP, called *fast quorums* [Lam06b].

When clients have direct access to a local replica, the recently developed consensus protocol Mencius [MJM08] has been shown to outperform CP. However often, clients and servers are not co-located. When clients are using a remote service replicated for disaster tolerance, none of the mentioned protocols has the final say.

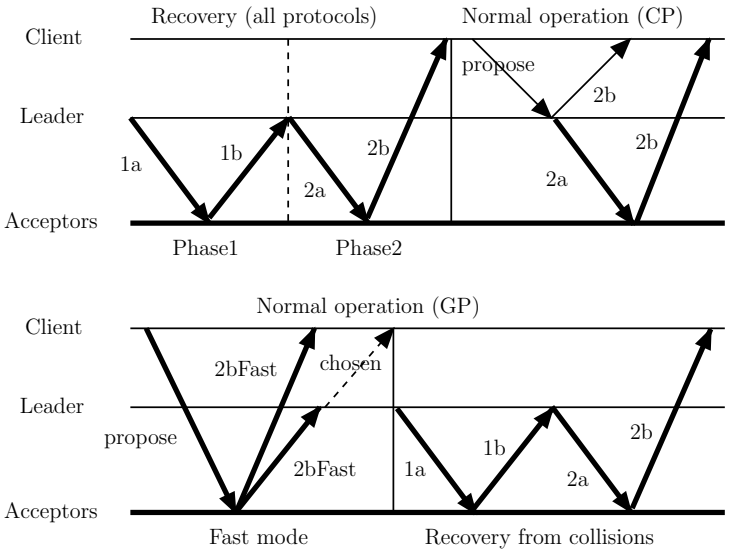


Figure 4.1: Paxos message patterns

4.1.1 Contributions

We present Hybrid Paxos (HP), a generalized consensus implementation that essentially extends CP with an additional "fast mode", enabling learning in two message delays in the absence of collisions. In presence of collisions, when GP takes six message delays, HP requires only three message delays. These latencies are optimal [Lam06b] and they are attained using a linear number of messages and the optimal number of $2f + 1$ servers, where f is the bound on crash-failures. Compared to Mencius, HP uses weaker synchrony assumptions, resulting in higher availability in WANs.

We show for the first time that generalized consensus can be used in practice to build efficient replicated services. The key to efficiency is that fast learning must not impact the bottleneck, which in CP is the leader. Additional messages in HP are exchanged only between clients (which are both proposers and learners) and acceptors. Thus, HP exploits the relative underutilization of the acceptors and offers a better latency up to 70% of the peak throughput of CP.

In addition, fast learning is enabled only if spare capacity is available. This is done by adaptively switching it on and off based on the load. Our evaluation using Emulab [WLS+03] shows that the latency of HP reaches the theoretical minimum. In the presence of collisions and with increasing load, HP behaves like CP.

4.1.2 No Clear Winner with CP and GP

We argue that the quorum size has a significant performance impact by showing that even in the absence of collisions, CP can outperform GP.

We have sampled delays among Planetlab [BBC+04] nodes and used them to simulate the normal operation (best case) of CP and GP in four different WAN settings (Table 4.1). The client distribution is as follows: 56% are located in the US, 38% in Europe, and 6% in Asia. All topologies use 11 servers. GP requires a fast quorum (9 servers), while CP only requires a (majority) quorum. The simulation results in Figure 4.2 suggest that: (a) in many settings, some clients are better off using GP and others prefer CP and (b) the crash of a single server can turn a setting beneficial for GP into one beneficial only for CP. Thus, there exist practical settings where neither of the two protocols always outperforms the other.

Table 4.1: WAN server layout (11 servers)

Topology	**Europe**	**World**	**CLUS-5(4)**
Leader Site	Hungary	Japan	Switzerland
Backups	Europe	Global	Europe
Clusters	No	No	1 with 5 (4) nodes, 3 with 2 nodes
Quorums	6 servers for CP, 9 servers for GP		

In the *Europe* setting, servers are located at 11 different sites in Europe. For most clients, the distance between them and the servers is close to uniform. Thus, the GP pattern leads to good results: 28% of the clients observe that GP is at least 10% faster than CP, and 10% of the clients even observe a 20% improvement. However, 12% of the clients find that CP is 10% better, as they do not have good connectivity with three additional servers required by GP. This supports observation (a).

The *World* setting models a world-wide setting in which acceptors are spread over the US, Europe and Asia, and the leader is located in Asia. In this scenario the advantage of the GP pattern is even clearer: 75% of the clients observe a 10% improvement over CP, and for 20% of the clients the improvement is $> 35\%$. The reason is the additional message delay to reach the leader, which is large for most of the clients. However, there are some clients which prefer CP: 6% find it to be 20% more efficient than GP, supporting observation (a). This is essentially the fraction of Asian clients which can quickly reach the leader.

The *CLUS* topology considers the case when servers are clustered at four different sites in Europe, providing cheap disaster tolerance. The only

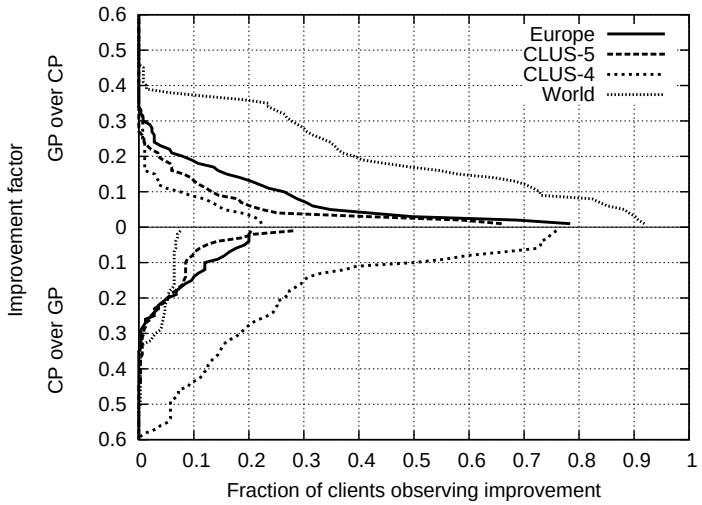


Figure 4.2: Improvement factor of GP over CP and vice versa

distinction between the *CLUS-5* and *CLUS-4* is that in the latter, one node in the largest cluster is crashed (Table 4.1). Before the crash, a fast quorum (9 servers) can be reached by contacting three sites. Note that for 13% of the clients, GP outperforms CP by at least 10%. Two sites are sufficient for CP to sample a quorum (6 servers). However, after the crash GP accesses *all* sites. This results in a shift of the performance profile, with CP dominating GP for 50% of the clients, supporting observation (b).

4.2 Model

We now give a brief summary of the model used herein, which is in line with the definitions given earlier in Chapter 2. We consider a message-passing system consisting of n servers and any number of clients. For simplicity, we assume reliable FIFO channels which can be easily implemented on top of reliable channels using standard techniques [AW98]. Further we consider a crash-stop model in which clients and servers fail only by crashing and nonfaulty servers never crash [1]. We allow any number of clients to crash, however we assume that at most $f < \lfloor n/2 \rfloor$ servers fail, which is necessary to solve consensus [CT96]. The system is asynchronous, with no bounds on message delay or processor speed. Each server has access to a failure detector Ω, that eventually outputs at all servers the same nonfaulty server.

[1] However, the algorithm can be extended to a model in which crashed nodes may recover [ACT98] and links are fair-lossy [BCBT96].

4.3 Generalized Consensus and Paxos

In this section we review Paxos and describe it as consensus on a growing command history [Lam05]. When needed, we differentiate between GP and CP. This description serves as a basis for HP which is introduced in Section 4.4.

In the client/server system that we consider, clients are both proposers and learners. The servers are acceptors and cooperate to choose a single command history. Acceptors query the Ω failure detector that elects a leader among them. Safety is guaranteed even if no leader or multiple leaders are elected, but a unique leader is required to ensure progress. Paxos operates on a set of round numbers. The round numbers are partitioned among the potential leaders such that each leader has its disjoint set of round numbers.

As mentioned in the introduction of this chapter, Paxos assumes predefined sets of acceptors called *quorums*. The requirement for CP is that any two quorums intersect. GP requires larger quorums, called *fast quorums*, and the requirement is that the intersection of any fast quorum FQ and any quorum is larger than $n - |FQ|$.

Following the Paxos protocol description [Lam98], we divide the acceptor and leader actions in Phase 1 and Phase 2 actions. Phase 1 actions are executed when a new round is started (e.g. after a leader crash). Phase 2 actions (1) complete choosing all the histories that failed to be chosen in an earlier round and (2) they are repeatedly executed during normal operation.

We now describe the algorithms' actions below (see also Fig. 4.1 as an illustration). Note that the focus lies on consensus, and therefore the execution of commands is omitted from the description.

Phase1: Start a new round

(1a) Leader l picks a new round number r from its set of round numbers and sends a $\langle \text{“}1a\text{”}, r \rangle$ message to all acceptors.

(1b) When acceptor a receives a $\langle \text{“}1a\text{”}, r \rangle$ message from leader l, if it has not yet received a message with a higher round number, then it replies with a $\langle \text{“}1b\text{”}, r, \ldots \rangle$[2] message. We say that a has moved to round r and considers r as its current round from now on. Moreover, a stops accepting proposals from clients.

If a has already received a message with round number $r' > r$ then it sends a message to the leader, indicating that it is ignoring the $\langle \text{“}1a\text{”}, r \rangle$

[2]Note that “...” will be replaced by protocol specific information later.

message. (Upon receiving such a message the leader performs step (1a) with a round number $> r'$ if it still believes to be the leader.)

Phase2: Complete earlier rounds

(2a) If leader l has received $\langle \text{“}1b\text{”}, r, \dots \rangle$ messages from a quorum of acceptors, then it sends a $\langle \text{“}2a\text{”}, r, h \rangle$ message to the acceptors where h is the history that has been determined from the received “$1b$” messages. Further, the leader adopts h as the history currently chosen. The rule of picking h depends on the type of protocol and is described later (see Sections 4.3.1 and 4.4.2).

(2b) If acceptor a receives a $\langle \text{“}2a\text{”}, r, h \rangle$ message in its current round r (i.e. it has not yet received any message with a higher round number), it stores h as the accepted history and sends a $\langle \text{“}2b\text{”}, r, h \rangle$ message to every learner. Next, a starts accepting proposals from clients.

(Learn) If a learner receives identical $\langle \text{“}2b\text{”}, r, h \bullet c \rangle$ messages from a quorum, then it learns that history $h \bullet c$ is chosen.

Normal operation CP

(Propose c) Client cl sends a $\langle \text{“}propose\text{”}, c \rangle$ message to the leader.

(2aCP) When the leader receives a $\langle \text{“}propose\text{”}, c \rangle$ message from client cl, it appends command c to h and sends a $\langle \text{“}2a\text{”}, r, h \bullet c \rangle$ message to the acceptors.

(2bCP) If acceptor a receives a $\langle \text{“}2a\text{”}, r, h \bullet c \rangle$ message from the leader in its current round r (i.e. it has not yet received any message with a higher round number), then it accepts $h \bullet c$ and sends a $\langle \text{“}2b\text{”}, r, h \bullet c \rangle$ message to all learners. Learning is done as described in the (Learn) step.

Normal operation GP

(ProposeGP c) Client cl sends $\langle \text{“}propose\text{”}, c \rangle$ messages to the acceptors.

(2bFast) If acceptor a receives a $\langle \text{“}propose\text{”}, c \rangle$ message from client cl, then a appends c to its command history h and sends $\langle \text{“}2bFast\text{”}, r, h \bullet c \rangle$ messages to the learners and to the leader.

(Collision Handling) If the leader receives identical $\langle \text{“}2bFast\text{”}, r, h \bullet c \rangle$ messages from a fast quorum, it indicates to the learners that $h \bullet c$ is chosen by sending $\langle \text{“}chosen\text{”}, r, h \bullet c \rangle$ messages to the learners. Else, the leader initiates collision recovery, which entails starting a new round (Phase 1) and recovering from earlier rounds (Phase 2).

(Fast Learn) If a learner receives identical $\langle \text{“}2bFast\text{”}, r, h \bullet c \rangle$ messages from a fast quorum, or equivalently a $\langle \text{“}chosen\text{”}, r, h \bullet c \rangle$ message then it learns that $h \bullet c$ is chosen. “Slow” learning is done as in (Learn).

4.3.1 The rule of picking a history

We now explain the core of the Paxos protocol and why it satisfies Consistency. For this purpose, we now informally describe the rule of picking a history based on the $\langle \text{“}1b\text{”}, r, \ldots \rangle$ messages received by the leader in step (2a). A formal and complete treatment appears in an earlier work by Lamport [Lam05].

Invariant Paxos maintains the following invariant for safety: if a history h is chosen in round r and a history h' is chosen in a higher numbered round, then $h \sqsubseteq h'$. Intuitively, Consistency follows from this invariant and the fact that once a quorum of acceptors has joined a higher numbered round, no history can be chosen in previous rounds anymore.

Pick classic In CP, if a history h has been chosen then learning implies that a quorum of acceptors has accepted h. In step (1b), each acceptor reports the history it has accepted. By the quorum intersection property, at least one acceptor reports h. The picking rule is to select the *lub* among the reported histories. Is is not difficult to see that $h \sqsubseteq lub$.

Pick fast In GP, if a history h has been chosen, then fast learning implies that a fast quorum FQ has accepted h. In step (1b), each acceptor reports the history it has accepted. Let Q be the set of all reported histories collected by the leader in step (2a). By the intersection property of FQ with a quorum, Q contains at least $n - |FQ| + 1$ (possibly incompatible) extensions of h and at most $n - |FQ|$ histories which are not extensions of h. Hence, there is a majority subset $M \subseteq Q$ containing the extensions of h. The goal is now to find a history which is an extension of h using this knowledge. First, the *glb* is computed for every majority subset $M \subseteq Q$. As all majorities intersect, the *glb*s are pairwise compatible. Next, the leader picks the *lub* of these *glb*s. Note that one of the *glb*s is an extension of h, and therefore $h \sqsubseteq lub$.

4.4 The Hybrid Paxos Protocol

As mentioned in the introduction of this chapter, HP is essentially CP with an integrated "fast mode", enabling fast learning in the absence of collisions. Therefore, the progress property of HP is inherited from CP. Hence, throughout the section, we will focus on safety.

The roles played by clients and servers and their interaction are the same as in Paxos (see Section 4.3). Phase 1 and 2 do not change and so they are as depicted in Figure 4.1. Figure 4.3 illustrates the message pattern of HP during normal operation.

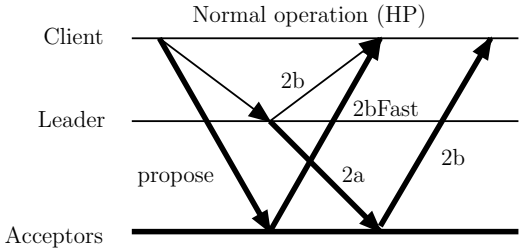


Figure 4.3: HP message pattern

4.4.1 Overview

We now briefly summarize the main differences between HP and Paxos.

First, fast learning is refined to accommodate that learning and fast learning are done in parallel, such that a learner can learn the quickest outcome. A naïve composition would fail, as two incompatible histories could be learned in the (Fast Learn) and (Learn) steps. We prevent this problem by replacing fast learning with *hybrid learning*. The idea of hybrid learning is that a learner waits for a fast quorum of identical "$2bFast$" and "$2b$" messages, of which at least one is of type "$2b$". Note that hybrid learning is fast because the leader is an acceptor (see Figure 4.3).

Secondly, the rule of picking a history in step (2a) is extended accordingly. In HP, each acceptor keeps two separate histories, a *classic* history updated by "$2a$" messages and a *fast* history updated by client proposals. Both histories are reported to the leader in step (1b). The leader applies the picking rules as described in Section 4.3.1 to each type separately. Resulting are two histories h and fh, where fh is determined by the fast histories. The final history is determined as the *lub* of h and the largest prefix of fh which is compatible with h.

4.4.2 The Protocol

We now describe the actions of the HP protocol during normal operation. The focus lies on highlighting the difference to CP. A complete description in pseudocode and proofs are given in Section 4.6.

Phase 1 and 2 actions as well as actions (2aCP), (2bCP) and (Learn) are actions of the HP protocol. Since they do not change, they are not listed below.

Normal Operation

(ProposeHP c) Client cl executes the actions (Propose c) and (ProposeGP c).

(2bFastHP) If acceptor a receives a $\langle$"*propose*"$, c\rangle$ message from client cl, then a appends c to the local fast history fh and sends $\langle$"$2bFast$"$, r, fh \bullet c\rangle$ messages to the learners. (Note that the difference to action (2bFast) is that c is appended to the fast history, and that no "$2bFast$" messages are sent to the leader).

(Hybrid Learn) If a learner receives identical $\langle$"$2bFast$"$, r, fh \bullet c\rangle$ messages from a fast quorum *and* one $\langle$"$2b$"$, r, h \bullet c\rangle$ message *and* $h = fh$ then it learns that $h \bullet c$ is chosen. "Slow" learning is done as in (Learn).

The rule of picking a history

We now explain the rule of picking a history in HP just as we did for Paxos. We will widely reuse the steps in Section 4.3.1 and refer to them when needed.

In HP, the $\langle$"$1b$"$, r, \dots\rangle$ messages report two separate, accepted histories, the (classic) history and the fast history. The leader uses the reported (classic) histories to pick a history h as described in **Pick classic**. Next, the leader uses the reported fast histories to pick a history fh as described in **Pick fast**. Note that each of the histories satisfies the invariant. History h is an extension of any history learned in the (Learn) step and fh is an extension of any fast history learned in the (Hybrid Learn) step.

Pick hybrid We now explain the key difference between HP and Paxos. To be safe, ideally we would pick the *lub* of h and fh and initialize the acceptors with *lub* in phase (2a). However, if h and fh are incompatible, then their *lub* is undefined. Therefore, the idea is (1) to determine the largest prefix pfh of fh which is compatible with h and (2) to pick the *lub* of pfh and h. This

would be safe only if we can guarantee that any history lh learned in step (Hybrid Learn) is a prefix of pfh.

We will now argue that this is the case. We know that lh is a prefix of fh. By the choice of pfh, all prefixes of fh which are compatible with h are also prefixes of pfh. Thus, it suffices to show that lh is compatible with h. Hybrid learning implies that some acceptor has accepted lh as (classic) history. Thus, lh is a prefix of some history sent by the leader in a "2a" message. Clearly, this holds for h too. Any two histories sent by the leader (of the same round) are prefixes of each other. So, if max is the largest of the two histories, then max is a common extension of lh and h. Thus, lh and h are compatible.

Implementation Considerations

Now that we have argued about the correctness (safety) of HP, in this section we describe how HP can be tweaked to be practical. We have identified a set of optimizations and listed them below.

O1: The leader does not have to send the entire history $h \bullet c$ in step (2aCP), it suffices to send c. When an acceptor receives c in a "2a" message, the FIFO property implies that it has already received h.

If the state machine is implemented at the servers, then there is no reason to send the entire history to the clients. All a client needs to learn is **(a)** the execution result of its last issued command and **(b)** that the history producing the result is chosen.

O2: The solution to **(a)** is to have the servers speculatively execute commands. Specifically, when the leader receives a proposal from a client, it immediately executes the command and includes the result in the "2b" message it sends back to the client. Speculation at the leader avoids rollbacks and history replays during normal operation.

O3: In order to attain **(b)** without sending the history, we replace the histories in the "2b" and "2$bFast$" messages with *history digests*. Two history digests are equal iff the corresponding histories are equal. Thus, in the (Learn) and (Hybrid Learn) step, clients check history digests for equality. Intuitively, a history digest function takes as arguments a history h and a command c contained in that history. It then computes the smallest prefix of h containing c and returns the digest thereof. We refer the reader to Appendix A for an incremental digest implementation based on hashing.

O4: If the classic and fast histories diverge during normal operation, the protocol as described above prevents hybrid learning. A simple solution would be to periodically start a new round. However, this imposes a considerable overhead. Therefore, the idea is to have each acceptor locally *align* the fast history fh to the classic history h as follows. Periodically, fh is replaced with the *lub* of h and the largest prefix of fh that is compatible with h. We know from Section 4.4.2 that this is safe.

O5: We have optimized HP to adapt to a changing workload. Specifically, HP uses a double threshold (T_{high}, T_{low}) such that when the load increases above T_{high} (resp. decreases below T_{low}) the fast mode is switched off (resp. switched on). Changes in the load are monitored by the leader, who is counting proposals per time unit. The leader simply tells the clients (in "2*b*" messages) to stop (respectively start) sending commands to the acceptors.

4.4.3 Discussion

In this section, we provide a comparison of HP and GP in terms of their performance during normal operation. We argue that the cost of collision recovery in GP outweighs the gain obtained from fast learning. In the original Fast Paxos paper [Lam06a] says: "If collisions are very rare, then starting a new round might be best. If collisions are too frequent, then classic Paxos might be better than Fast Paxos."

The latency of HP and GP equals two message delays in the absence of collisions. In the presence of collisions, HP requires three message delays and GP requires *six*. Hence, if GP is recovering from collisions only 25% of the time, then there is no (average) gain from fast learning.

The message complexity of HP is $4n$ messages per request. GP requires $3n+1$ messages in the absence of collisions and $(6+l)n$ messages otherwise, where l is the number of learners. If we consider that $l = n \geqslant 3$, and GP is recovering from collisions only 12% of the time, then GP has a higher (average) message complexity. Even without collisions, in GP the leader collects "2*bFast*" messages and checks if collisions occurred, thus becoming the bottleneck.

Our experiments with HP have revealed that with increasing load, the collision rate is growing faster than the server capacity utilization rate. For instance, we have observed that the servers are still underutilized when the rate of hybrid learning drops under 50% (with 99% commutable commands).

In this situation, GP would spend > 50% of the time recovering from collisions, thus performing poorly compared to HP.

4.5 Evaluation

This section explores the performance characteristics of HP and compares it to existing approaches. As argued in Section 4.4.3 above, we expect HP to outperform GP in most situations, and therefore we omit a direct comparison. We substantiate our claim by showing that HP's latency often attains the theoretical minimum. We compare HP to CP and show that it performs significantly better under low to medium load *and* equally well under high load. Where appropriate, we also compare HP to Mencius [MJM08].

4.5.1 Experimental Settings

We have implemented a simple banking system in which multiple clients share a bank account. Clients can deposit or withdraw money. The state consists of the balance of the shared account, and clients can issue *withdraw* or *deposit* commands. Executing *withdraw* \$100 subtracts \$100 in a state with at least \$100 and produces \$100 as output. Executing *deposit* \$20 adds \$20 and produces OK as output. Note that any two *deposit* commands are commutable because executing them in either order has the same effect. However, when one of the two operations is a *withdraw*, the order matters.

A scenario is modeled in which clients frequently deposit small amounts of money and less frequently withdraw larger amounts. Where the rate of *withdraw* commands matters, we use "HP-x" to denote runs of the HP protocol, where on average, one out of x commands is a *withdraw*. We use "CP3" (respectively "CP4") to denote the specific CP protocol where a command can be learned by a client after three (respectively four) message delays; CP4 relates to CP without speculation.

We ran experiments in the Emulab testbed [WLS+03] and we implemented all protocols in Java using the Neko [UDS01] framework. The protocols are evaluated in a system with five servers ($f = 2$) except for fault-scalability, where the number of servers is scaled up to 21 ($f = 10$).

Client and server nodes are connected by links with a one-way delay of $20ms$ and a bandwidth of 100 Mbps. The chosen delays are comparable to the "Europe" WAN setting analyzed in Section 4.1.2. The chosen network bandwidth models modern high-end WAN links such Geant2 [Gea]. Server nodes are 600 Mhz PCs with 256 MB memory running Fedora 6.

4.5.2 Latency

Figure 4.4 shows the average latency of HP under low and medium load as the rate of *withdraw* operations is varied between 0% and 100%. Note that the withdraw rate corresponds to the probability of collisions. For a *withdraw* rate of 0.5% and load offered by 100 clients, HP has a 32% lower latency than CP3. This is close to the theoretical minimum. For a *withdraw* rate of 100% and load offered by 10 clients (between 0.1 and 0.2 Kops), HP still features a latency of 20% below the optimum of CP3.

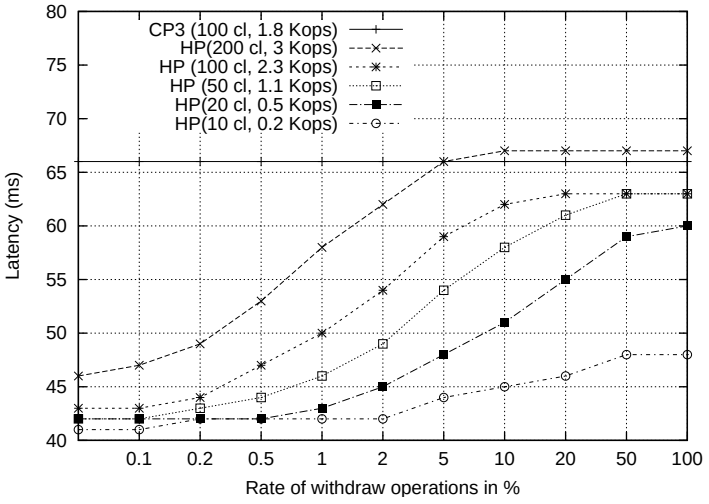


Figure 4.4: Latency versus *withdraw* rate

Figure 4.5 compares the latency and throughput of HP-500, CP3, CP4 with and without request batching (of 20 commands) as we vary the offered load. As illustrated, under low load, batching at the leader increases the latency of CP4 and CP3 but not that of HP because most commands ($> 90\%$) are chosen in the fast mode. On the other hand, batching increases peak throughput. In fact, with batching all protocols converge to the same peak throughput. Starting from a throughput of 6 Kops, the curves of HP and CP3 coincide because the fast mode is switched off.

Figures 4.6 and 4.7 illustrate the effectiveness of adaptive switching by means of a dynamic workload. The workload is organized as follows: 50% of the commands are sent under moderate load generated by 100 clients and 50% under high load generated by 1000 clients between $t = 68$ and $t = 98$. Figure 4.6 compares the average latency of HP with that of CP3 and CP4. We have measured the latency of HP with and without adaptive switching. The latter is referred to as nonadaptive. Figure 4.6 clearly shows that

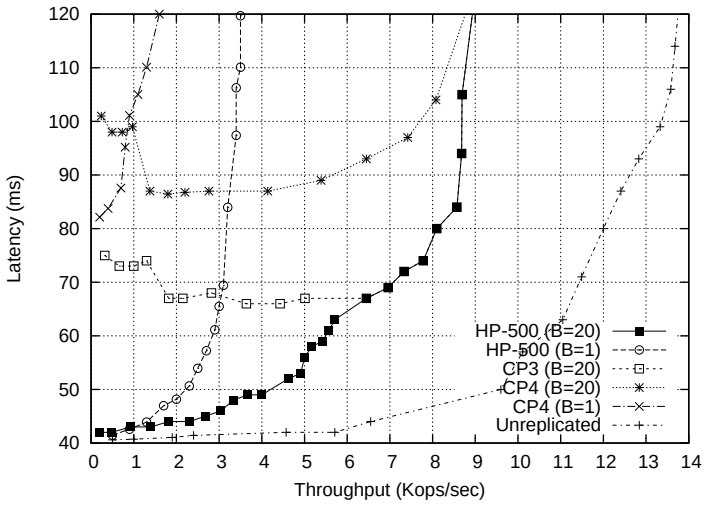


Figure 4.5: Latency versus throughput

during the high load burst, the nonadaptive version of HP performs worst among all protocols. The explanation is the following. Batching offloads the leader and the acceptors become the bottleneck nodes because they process more messages. In contrast, the adaptive version shows a short spike after the load burst starts and a short tail after the burst ends. These can be attributed to conservative thresholds. Overall, the adaptive version of HP features the minimum latency of all protocols. Figure 4.7 compares the cumulative latency distribution of the four protocols under the same workload and confirms that adaptive HP performs best both under moderate and high load.

Latency under Network Variance

So far, the experiments have been conducted in a setting in which the network is always timely. We now add a Pareto distribution to each link using the NetEm [Hem05] utility. The one-way network delay now varies between $20ms$ and $60ms$. Pareto is a heavy tailed distribution, which models the fact that wide-area links are usually timely (e.g. 80% of the time) but can present high latency occasionally.

Figure 4.8 compares the latency and throughput of HP-500, CP3, CP4 with batching as we vary the offered load. The trends are the same as in a situation with no network variance. An important point is that all protocols have lower peak throughput, including the unreplicated system. High variance results in packet reordering and packet retransmission at the

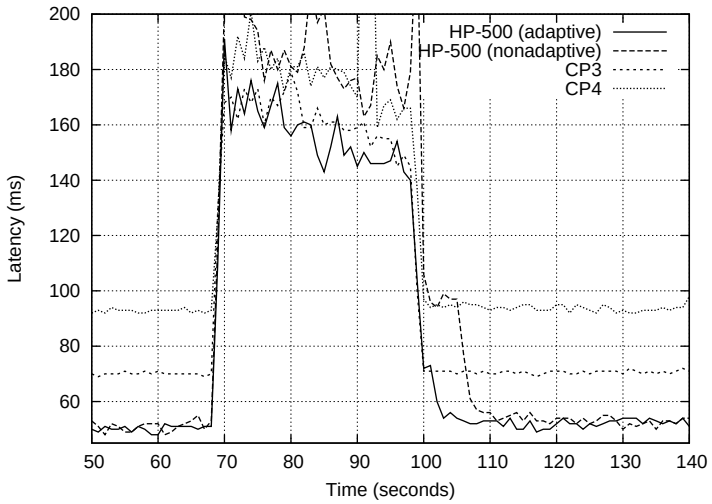


Figure 4.6: Average latency under a changing load (B = 20)

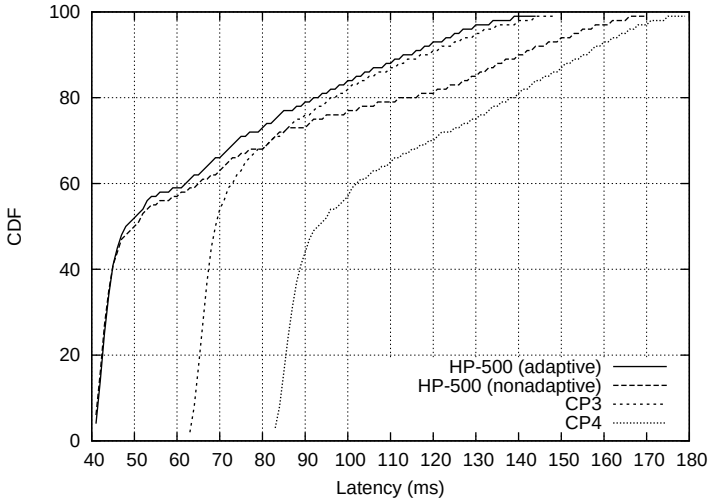


Figure 4.7: Latency CDF under a changing load (B = 20)

transport protocol level (TCP), causing additional load on the bottleneck node. HP outperforms CP3 up to 60% of the peak throughput. Up to a throughput of 1 Kops, HP and the unreplicated system have comparable latencies. The performance profile of HP is somewhat surprising because with high network variance, the likelihood of collisions increases. E.g., under a link variance of $40ms$, if two interfering commands are sent within $40ms$ from each other, they might be accepted in different orders.

Figure 4.9 supports the above observation showing that under network

Figure 4.8: Latency versus throughput (B = 20)

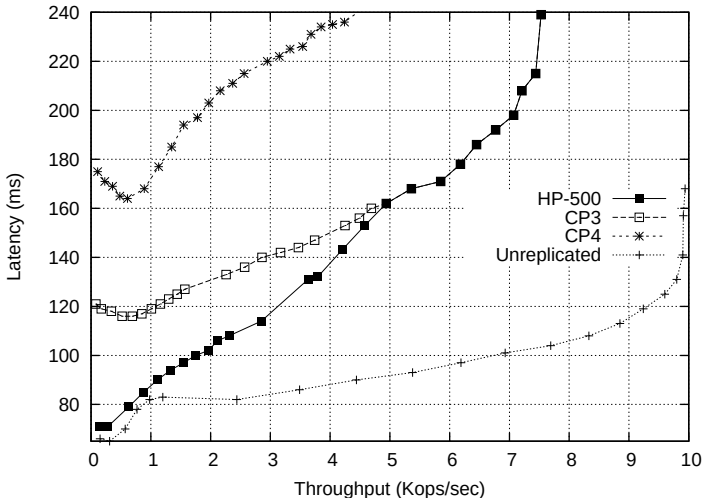
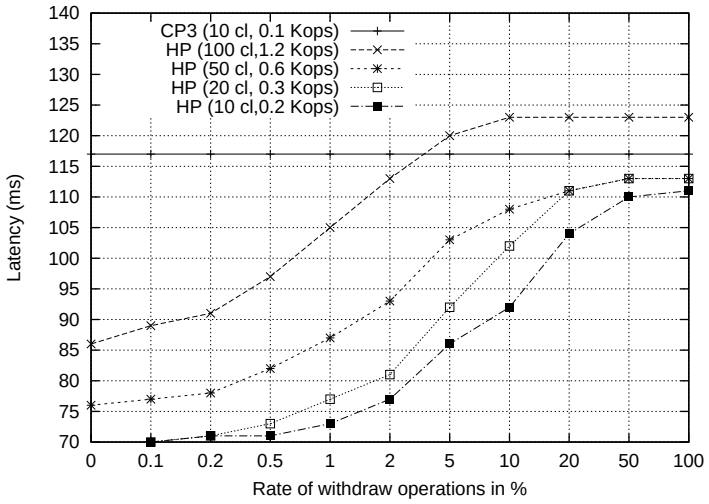


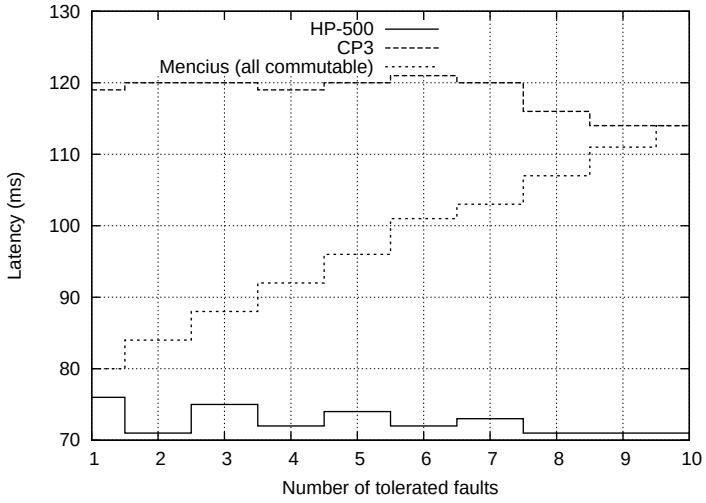
Figure 4.9: Latency vs. withdraw rate



variance, the latency of HP converges much faster to that of CP3. Nevertheless, under low load and a small fraction of withdraws, HP shows a latency improvement of up to 40% over CP3, which is more than the theoretical maximum latency reduction of 1/3. An explanation could be that the longer it takes to run an instance of a protocol, the more likely it is to depend on a slow link in the critical path. In this particular case, this effect adds to the latency of CP3 and explains the measured latency difference.

Figure 4.10 compares the latency of HP-500, CP3 and Mencius [MJM08] as more servers are added to the system. We are simulating a lightly loaded

Figure 4.10: Latency as f increases (20 clients)



scenario with 20 clients. With Mencius, a server has to wait for all other servers to skip or to propose a command. For a fair comparison, all commands are commutable and thus Mencius can commit in only one message roundtrip after receiving a reply from all servers, which is optimal. Mencius' dependency on slow links grows as more servers are added and therefore its latency increases. In contrast, the latency of HP and CP3 remains roughly constant (CP3's latency even drops) because they wait for the fastest quorum. These results suggest that the latency of CP and HP strongly depends on how large is the fraction of nodes that form a quorum. We observe that the latency oscillates (and even drops) with this fraction.

4.5.3 Throughput

We now show that the lower latency of HP does not come at the cost of lower throughput compared to CP.

Figure 4.11 shows the throughput of HP-500, CP3 and CP4 with and without batching as the number of clients increases. All protocols scale equally well when batching is used; CP4 without batching scales poorly. Figure 4.12 compares the peak throughput of HP (that equals CP3), CP4 and Mencius as the number of faulty servers tolerated increases. The throughput of Mencius is scaled down from 3GHz machines to ours (600MHz) using a factor of 1/4. The results show that HP outperforms all other protocols except in the case of $f = 2$ with batching, when its peak throughput is comparable to CP4. The fault scalability of HP is superior to that of CP4

with and without batching. For $f = 10$ with batching, HP features 73% of the peak throughput for $f = 2$. In contrast, CP4's peak throughput drops down to 50%.

Figure 4.11: Throughput as the number of clients increases

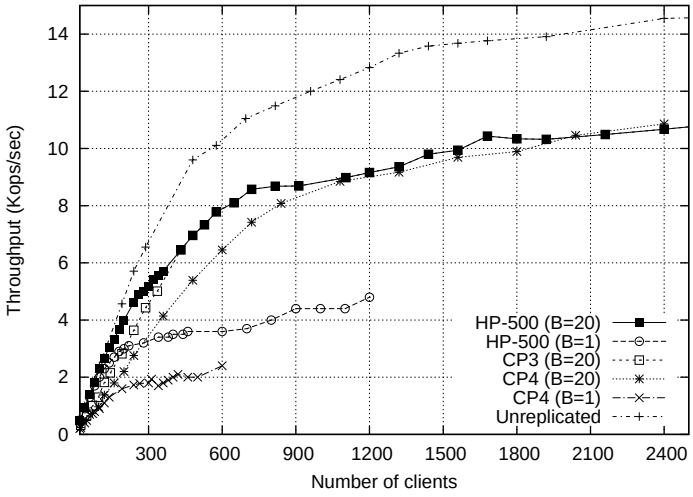
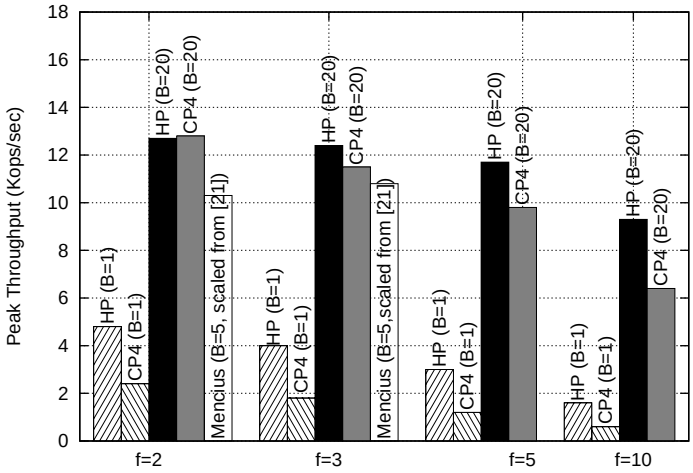


Figure 4.12: Throughput as f increases



4.6 Proof of Correctness

In this section we show that the HP algorithm depicted in Figures 4.13, 4.14, and 4.15 solves the generalized consensus problem previously defined in

```
1  k = 0, 1, 2, ...                          /* round index                    */
2  rnd, r_Max integer, initially 0           /* round number                   */
3  chist command history, initially ⊥        /* chosen history                 */
4  ch, fh command histories                  /* results of pick history        */
5  pfh command history, initially ⊥          /* largest prefix of fh
       compatible with ch                                                      */
6  recover boolean, initially true           /* current phase                  */
7  1bSet set of messages, initially empty    /* ''1b'' messages                */
8  leader boolean, initially false           /* leader process                 */

9  upon LEADER do
10    leader ← true
11    1bSet ← ∅
12    recover ← true
13    rnd ← k · n + l
14    send ⟨"1a", rnd⟩ to all acceptors
15 upon NOTLEADER do
16    leader ← false
17 upon receive ⟨"1b", rnd, vr, fh, ch⟩ from j do
18    1bSet ← 1bSet ∪ {(vr, fh, ch, j)}
19 upon |1bSet| ⩾ n − f ∧ recover do
20    r_Max ← Max({r | (r, fh, ch, j) ∈ 1bSet})
21    1bSet ← {(vr, fh, ch, j) ∈ 1bSet | vr = r_Max}
22    ch ← pickClassicHistory(1bSet)
23    fh ← pickFastHistory(1bSet)
24    pfh ← ⨆{pref ⊑ fh : pref compatible with ch}
25    chist ← pfh ⊔ ch
26    send ⟨"2aStart", rnd, chist⟩ to all acceptors
27    recover ← false
28 upon receive ⟨"nack", r⟩ : leader ∧ r > rnd do
29    k ← ⌊r/n⌋ + 1
30    trigger LEADER
31 upon receive ⟨"propose", c⟩ from client cl : leader ∧ ¬recover do
32    if c ∉ chist then
33       chist ← chist • c
34       send ⟨"2a", rnd, c, cl⟩ to all acceptors
35    send ⟨"2b", rnd, c, historyDigest(c, chist)⟩ to client cl
```

Figure 4.13: Algorithm of Leader l

Section 2.2.6. We do this by showing that HP satisfies *Conservatism, Consistency, Progress* and *Nontriviality*. We first give some helpful definitions

```
1  rnd integer, initially 0      /* round number, determines leader    */
2  fhist command history, initially ⊥      /* fast history        */
3  chist command history, initially ⊥     /* classic history      */
4  pfh command history, initially ⊥        /* largest prefix of fhist
       compatible with chist                                       */
5  recover boolean, initially true  /* accept proposals iff ¬recover */
6  upon receive ⟨"1a", r⟩ : r > rnd  do
7       send ⟨"1b", r, rnd, chist, fhist⟩ to leader
8       recover ← true
9       rnd ← r
10 upon receive ⟨∗, r, ∗⟩ : r < rnd  do
11      send ⟨"nack", rnd⟩ to r(mod n)
12 upon receive ⟨"2aStart", r, rh⟩ : r ⩾ rnd  do
13      rnd ← r
14      fhist ← chist ← rh
15      recover ← false
16 upon receive ⟨"2a", r, c, cl⟩ : r ⩾ rnd  do
17      rnd ← r
18      chist ← chist • c
19      pfh ← ⨆{pref ⊑ fhist : pref compatible with chist}
20      fhist ← chist ⊔ pfh
21      send ⟨"2b", rnd, c, historyDigest(c, chist)⟩ to client cl
22 upon receive ⟨"propose", c⟩ from client cl ∧ ¬recover  do
23      if c ∈ chist then
24           send ⟨"2b", rnd, c, historyDigest(c, chist)⟩ to client cl
25      if c ∉ fhist then
26           fhist ← fhist • c
27           send ⟨"2bFast", rnd, c, historyDigest(c, fhist)⟩ to client cl
```

Figure 4.14: Algorithm of the Acceptors

and prove a set of auxiliary lemmas.

Definition 5 (Accepted in round r)**.** *A command history h is accepted as classic history (respectively as fast history) in round r by acceptor j iff $chist_j \sqsupseteq h \ \wedge \ r_j = r \ \wedge \ \neg recover_j$ (respectively iff $fhist_j \sqsupseteq h \ \wedge \ r_j = r \wedge \neg recover_j$). A command history h is accepted in round r iff h is accepted as classic or fast history in round r by some acceptor.*

```
1  lc command, initially ⊥            /* last command issued          */
2  2bSet set of pairs (int, digest), initially ∅     /* ''2b'' msgs.   */
3  2bFastSet set of pairs (int, digest), init. ∅ /* ''2bFast'' msgs.    */
4  ref byte array, initially null      /* classic history digest used as
       reference                                                       */
5  pending boolean, initially false       /* pending operations       */
6  rnd integer, initially 0     /* round determining curr. leader     */
7  upon receive ⟨TYPE, r, c, hd⟩ from j : TYPE ∈ {"2b","2bFast" } ∧
   r ⩾ rnd ∧ c = lc ∧ pending  do
8      if r > rnd then
9          rnd ← r
10         2bFastSet ← 2bSet ← ∅
11         ref ← null
12     if TYPE = "2b" then
13         2bSet ← 2bSet ∪ {(j, hd)}
14         if ref = null then
15             ref ← hd
16     2bFastSet ← 2bFastSet ∪ {(j, hd)}
17     2bFastSet ← 2bFastSet\{(i, hd) ∈ 2bFastSet|hd ≠ ref}
18     if (|2bFastSet| ⩾ ⌈(n+f+1)/2⌉ ∧ |2bSet| ⩾ 1) ∨ (|2bSet| ⩾ ⌈(n+1)/2⌉)
       then
19         learn(lc)
20         2bFastSet ← 2bSet ← ∅
21         ref ← null
22         lc ← ⊥
23         pending ← false
24 upon propose(c) ∧¬pending  do
25     pending ← true
26     lc ← c
27     while pending do
28         send ⟨"propose", c⟩ to all servers
```

Figure 4.15: Algorithm of the Clients

```
1  ch ← ⨆{h|(vr, fh, h, j) ∈ 1bSet}
2  return ch
```

Figure 4.16: **Function** `pickClassicHistory`$(1bSet)$

```
1 Comp ← {(h, j)|(vr, h, ch, j) ∈ 1bSet}
2 if Comp is incompatible then
3       repeat
4          Comp ← Comp\{(h, i), (h′, j)|i ≠ j ∧ h incompatible with h′}
5          p ← h ⊓ h′
6          Comp ← Comp ∪ {(p, i), (p, j)}
7       until Comp is compatible
8  fh ← ⨆{h|(h, j) ∈ Comp}
9 return fh
```

Figure 4.17: **Function** `pickFastHistory`($1bSet$)

Definition 6 (Chosen in round r)**.** *A command history h is chosen in round r iff h is accepted in round r*

(Classic) *as classic history by a quorum Q of $\lceil \frac{n+1}{2} \rceil$ acceptors*

OR

(Fast) *(1) as fast history by a quorum FQ of $\lceil \frac{n+f+1}{2} \rceil$ acceptors and (2) as classic history by some acceptor.*

Lemma 9. *If any two classic histories h and h' are accepted in round r, then h and h' are compatible.*

Proof. By the FIFO property of the channels, if any two histories h and h' are accepted in round r, then both are prefixes of the history proposed by the leader of round r. Thus, h and h' are compatible. □

Lemma 10. *If any two histories h and h' are chosen in round r, then h and h' are compatible.*

Proof. If h and h' are chosen in round r, then by Definition 6, both are accepted as classic histories in round r. By Lemma 9, h and h' are compatible. □

Lemma 11. *If history h is chosen in round r and h is accepted by a quorum Q, then for every acceptor $j \in Q$ holds: if j accepts h' at any later time in round r, then h' is an extension of h.*

Proof. We treat the two cases when h is accepted as classic history (case A) and h is accepted as fast history (case B) separately.

Case A: Note that the classic history accepted by any acceptor j in round r is a prefix of the history accepted by j at any later time in r (Fig. 4.14, line 18). This also applies to classic histories that are chosen.

Case B: According to Definition 6, if h is chosen, then h is the prefix of some classic history accepted in round r. By Lemma 9, it follows that h is compatible with *every* classic history accepted in round r. Hence, for every $j \in Q$, it holds that $h \sqsubseteq fhist_j$ and h is compatible with $chist_j$. Therefore, $\bigsqcup\{pref \sqsubseteq fhist_j : pref \text{ is compatible with } chist_j\}$ is an extension of h. Thus, once h is chosen, $fhist_j$ is updated only with extensions of h (Fig. 4.14, lines 14, 20, 26). □

Lemma 12. *If h is chosen in round r and h' is accepted in round $r' > r$ then h' is an extension of h.*

Proof. We show the lemma by induction on round number k, where $r < k \leqslant r'$. For any k, let g be the history accepted by any acceptor in round k. Note that g is necessarily an extension of the history rh contained in the "$2aStart$" messages of round k. This is true because in round k (a) any acceptor j accepts client requests only after $fhist_j$ and $chist_j$ have been initialized with rh (Fig. 4.14, line 14) and (b) $rh \sqsubseteq chist_j \ \wedge \ rh \sqsubseteq fhist_j$ is invariant (Fig. 4.14, lines 18-20). Therefore, it is sufficient to show that rh, i.e., the history picked by the leader of round k is an extension of h.

Base step: let k be the lowest round number $k > r$ in which some acceptor accepts a history. We show that if h is chosen in round r, then the history picked by the leader of round k is an extension of h. We distinguish the cases when h is chosen (Case A) as classic history or (Case B) as fast history.

Case A implies that a quorum Q of $\lceil \frac{n+1}{2} \rceil$ acceptors have accepted h. Let Q' be the set of $n - f$ acceptors from which the leader of round k receives "$1b$" messages and let $R \subseteq Q'$ be the subset of all acceptors that have voted in round r. By the definition of R and the quorum intersection property, $Q \cap Q' \subseteq R \neq \varnothing$. By Lemma 11, some acceptor in R reports an extension of h and by Lemma 9, all classic histories reported by acceptors in R are compatible. Therefore, `pickClassicHistory` (Fig. 4.16) returns an extension of h. Finally, the recovered history is also an extension of h (Fig. 4.13, line 25).

Case B implies that a quorum FQ of $\lceil \frac{n+f+1}{2} \rceil$ acceptors has accepted h as fast history and h is the prefix of some accepted classic history. Let Q' be the set of $n - f$ acceptors from which the leader of round k receives "$1b$" messages. Further let $R \subseteq Q'$ be the set of acceptors that have voted in r

and let $M \subseteq R$ denote the set of acceptors that have accepted h in r. By Lemma 11, all acceptors in M report extensions of h. By the intersection property of FQ and Q', $|M| > |R \backslash M|$. Let $Comp$ denote the set of fast histories reported by the acceptors in R. It is not difficult to see hat after lines 3—7 in Fig. 4.17 are executed, $Comp$ contains (a) only compatible histories and (b) some history in $Comp$ is an extension of h. The latter is true because a majority of histories contained in $Comp$ are extensions of h. Therefore, fh, the history returned by `pickFastHistory` (Fig. 4.17) is an extension of h. It remains to show that the recovered history is an extension of h. Recall that the assumption that h is chosen implies that h is the prefix of some accepted classic history. By Lemma 9, h is compatible with every classic history and hence with ch, returned by `pickClassicHistory`. Thus, the recovered history $\bigsqcup\{pref \sqsubseteq fhp : pref$ compatible with $ch\}$ is an extension of h.

Induction step: we show that the Lemma is true for $k = r'$ under the assumption that it holds for all $r < k < r'$. Recall that it is sufficient to show that the history picked by the leader of round r' is an extension of h. Let Q' be the quorum of $n - f$ acceptors from which the leader of round r' collects "1b" messages. Further, let $R \subseteq Q'$ denote the subset of acceptors that have voted in round r_{Max} (Fig. 4.13, line 20). Note that by the quorum intersection property, $r_{Max} \geqslant r$. The case $r_{Max} = r$ is covered by the proof of the base step. Therefore, we only show the case $r_{Max} > r$. The induction hypothesis implies that every acceptor in R has accepted in round r_{Max} an extension of h as classic history. Since classic histories never shrink throughout a round, all histories reported by acceptors in R are extension of h. Further, by Lemma 9, they are compatible histories. Hence, the history picked by the leader of round r' is an extension h. □

Lemma 13 (Consistency)**.** *If any two histories h and h' are chosen, then h and h' are compatible.*

Proof. We assume without loss of generality that h is chosen in round r and h' is chosen in round $r' \geqslant r$. If $r = r'$, then by Lemma 10, h and h' are compatible. Else if $r' > r$, by Lemma 12 h' is an extension of h, and thus h and h' obviously are compatible. □

Lemma 14 (Progress)**.** *If a command c is proposed by a non-faulty client cl, then cl eventually learns a command history containing c.*

Proof. By the property of Ω, exactly one non-faulty leader ld is eventually elected forever. By the construction of Phase 1 (Fig. 4.13, line 28-30), ld eventually starts a round with a number that is greater than any other round number. Therefore, ld and a quorum Q of $\lceil \frac{n+1}{2} \rceil$ correct acceptors eventually join the highest round ever. We assume by contradiction that cl never learns a history containing c. This implies that cl keeps on resending c to all replicas forever. Thus, ld eventually sends a "2a" message containing c to all acceptors unless it has already did so. In any case, every acceptor in Q eventually accepts a classic history containing c. From this point on, every acceptor in Q that receives a "*propose*" message from cl sends a "2b" message to cl (Fig. 4.14, line 24). Thus, cl eventually collects identical "2b" messages from $\lceil \frac{n+1}{2} \rceil$ acceptors (Fig. 4.15, line 18) voting for histories that contain c. Hence, cl stops resending command c, a contradiction. □

Lemma 15. *If h and h' are classic histories accepted in some round r and both h and h' contain c, then $h \sqcap h'$ contains c.*

Proof. By Lemma 9, h and h' are compatible histories. By axiom CS4 for command structures [Lam05] (here called histories), the greatest lower bound $h \sqcap h'$ also contains c. □

Lemma 16. *If two command histories h and h' both contain command c and* `historyDigest`$(c, h) =$ `historyDigest`(c, h')*, then $h \sqcap h'$ contains c.*

Proof. Let H be any function with the following property: for any two values x and y it holds that $H(x) = H(y)$ if and only if $x = y$. Function `historyDigest`(c, h) returns a value $H(sp)$, where sp is defined as the smallest prefix of h containing c. Let sp' be the smallest prefix of h' containing c and $H(sp') =$ `historyDigest`(c, h'). Given that $H(sp)$ and $H(sp')$ are equal, by the property of H we have that $sp = sp'$. Since sp contains c and sp is a prefix of $h \sqcap h'$, we conclude that c is also contained in $h \sqcap h'$. □

Lemma 17 (Conservatism)**.** *If a client learns a command history h, then h is chosen.*

Proof. Command history h is learned in line 19, Fig. 4.15.

Case 1 (learning): a quorum Q of $\lceil \frac{n+1}{2} \rceil$ acceptors j have accepted classic history h_j containing command c, where c is the last command issued by the

client. By Definition 11, $\bigsqcap\{h_j | j \in Q\}$, the greatest common classic history accepted by every acceptor in Q, is chosen. We only have to show that the learned history h is a prefix of $\bigsqcap\{h_j | j \in Q\}$. By Lemma 15, $\bigsqcap\{h_j | j \in Q\}$ also contains command c. Since h is a prefix of any classic history containing c, and $\bigsqcap\{h_j | j \in Q\}$ is an extension of some classic history containing c, $\bigsqcap\{h_j | j \in Q\}$ is also an extension of h.

Case 2 (fast learning): a quorum FQ of $\lceil \frac{n+f+1}{2} \rceil$ acceptors j have replied with history digest $hd_j = \texttt{historyDigest}(c, h_j)$, where c is the last command issued by the client and h_j is the history accepted by j. Additionally, some h_j is a classic history. By Definition 11, $\bigsqcap\{h_j | j \in FQ\}$, the greatest common history accepted by every acceptor in FQ, is chosen. Therefore, it is sufficient to show that the learned history h is a prefix of $\bigsqcap\{h_j | j \in FQ\}$. Note that c is contained in h_j, otherwise j does not respond. Furthermore, for any two acceptors i and j it holds that $hd_i = hd_j$. Hence, by Lemma 16, $\bigsqcap\{h_j | j \in FQ\}$ contains command c. Since h is a prefix of any classic history containing c, and $\bigsqcap\{h_j | j \in FQ\}$ is an extension of some classic history, $\bigsqcap\{h_j | j \in FQ\}$ is also an extension of h. □

Lemma 18 (Nontriviality)**.** *Every chosen command is proposed by some client.*

Proof. Every chosen command is proposed by some client because no faulty acceptor can undetectably corrupt commands. □

Theorem 2. *Algorithm Hybrid Paxos in Figure 4.13, 4.14, and 4.15 solves generalized consensus on command histories.*

Proof. The theorem follows directly from Lemma 13, 14, 17 and 18. □

4.7 Summary of the Results

We have described Hybrid Paxos, a generalized consensus implementation featuring minimal latency and competitive peak throughput in most situations. The core idea of HP is to add fast learning to CP. HP is to our knowledge the first generalized consensus protocol that attains the optimal latency of two message delays in the absence of collisions, and three message delays in the common case. Moreover, HP exhibits optimal resilience and optimal messages. We have shown that generalized consensus is a practical approach to replication in WANs. Our experimental results demonstrate that HP outperforms state of the art protocols.

Chapter 5

Robust Amnesic Storage

In this chapter we consider robust implementations of regular read/write storage using a collection of $3t + k$ base objects, t of which can be subject to Byzantine failures. We focus on amnesic algorithms that store only a limited number of values in the base objects. In contrast, non-amnesic algorithms store an unbounded number of values, ultimately leading to space exhaustion. Lower bounds on the time complexity of read and write operations are currently met only by non-amnesic algorithms.

We show for the first time that amnesic algorithms can also meet these lower bounds. We do this by giving two amnesic constructions: for k = 1, we show that the lower bound of two communication rounds is also sufficient for every read operation to complete, and for $k = t + 1$ we show that one round is sufficient for every operation to complete. Thus, our contributions answer in the affirmative questions **Q3.1** and **Q3.2** raised in Section 1.2.2.

5.1 Introduction

Motivated by recent advances in the Storage Area Network (SAN) technology, and driven by the availability of cheap commodity disks, distributed storage has become a popular method to provide increased storage space, high availability and disaster tolerance.

We address the problem of efficiently implementing a reliable read/write distributed storage service from unreliable storage units (e.g. disks), a threshold of which might fail in a malicious manner. Fault-tolerant access to replicated remote data can easily become a performance bottleneck, especially for data-centric applications usually requiring frequent data access. Therefore, minimizing the time complexity of read and write operations is essential. In this chapter, we show how optimal time complexity can be achieved using

algorithms that are also space-efficient.

Much recent publications, and this work as well, focuses on *regular* registers where read operations never return outdated values. A regular register is deemed to return the last value written before the read was invoked, or one written concurrently with the read (see Section 2.3 for a formal definition). Regular registers are attractive because even under concurrency, they never return spurious values as sometimes done by the weaker class of safe registers (Section 2.3). Furthermore, they can be used, for instance, together with a failure detector to implement consensus [ACKM06].

The abstraction of a reliable storage is typically built by replicating the data over multiple unreliable distributed storage units called *base objects*. These can range from simple (low-level) read/write registers to more powerful base objects like *active disks* [CM05] that can perform some more sophisticated operations (e.g. an atomic read-modify-write). Taken to the extreme, base objects can also be implemented by full-fledged servers that execute more complex protocols and actively push data [MAD02].

We consider Byzantine-fault tolerant register constructions where a threshold $t < n/3$ of the base objects can fail by being *non-responsive* or by returning *arbitrary* values (i.e., NR-arbitrary [JCT98]), without assuming data authentication to limit the adversary.

Furthermore, we consider *wait-free* implementations where concurrent access to the base objects and client failures must not hamper the liveness of the algorithm. Wait-freedom is the strongest possible liveness property, stating that each client completes its operations independent of the progress and activity of other clients [Her91]. As already discussed in Section 2.3, algorithms that wait-free implement a regular register from Byzantine components are termed *robust* [CGK07].

An implementation of a reliable register requires the (client) processes accessing the register via a high-level operation to invoke multiple low-level operations on the base objects. In a distributed setting, each invocation of a low-level operation results in one *round* of communication from the client to the base object and back. The number of rounds needed to complete the high-level operation is used as a measure for the time complexity of the algorithm.

Robust algorithms that store only a limited number of written values in the base objects are desirable, yet difficult to design. Algorithms that satisfy this property are called *amnesic* [CGK07]. With amnesic algorithms, values previously stored are not permanently kept in storage but are eventually erased by a sequence of values written after them. Amnesic algorithms thus effectively eliminate the problem of space exhaustion raised by non-amnesic algorithms, which take the approach of storing the entire version history.

Therefore, the amnesic property captures an important aspect of the space requirements of a distributed storage implementation.

5.1.1 Previous and Related Work

Despite the importance of *amnesic and robust* distributed storage, most implementations to date are either *not* robust or *not* amnesic. While some relax wait-freedom and provide weaker termination guarantees instead [ACKM06, HGR07], others relax consistency and implement only the weaker safe semantics [JCT98, MR98, ACKM06, GV06]. Generally, when it comes to robustly accessing (unauthenticated) data, most algorithms store an unlimited number of values in the base objects [GWGR04, GV06, GV07]. Also in systems where base objects push messages to subscribed clients [MAD02, BD04, AAB07], the servers store every update until the corresponding message has been received by every non-faulty subscriber. Therefore, when the system is asynchronous, the servers might store an unbounded number of updates. A different approach is to assume a stronger model where data is self-verifying [MR98, CT06, LR06], typically based on digital signatures.

The existing robust and amnesic storage algorithms [GLV06, ACKM07] do not achieve the same time complexity as non-amnesic ones. Time complexity lower bounds have shown that protocols using the optimal number of $3t + 1$ base objects [MAD02] require at least *two rounds* to implement both read/write operations [GV06, ACKM06]. So far these bounds are met only by non-amnesic algorithms [GV07]. In fact, the only robust and amnesic algorithm with optimal resilience [GLV06] requires an unbounded number of read rounds in the worst case. For the $4t + 1$ case, the trivial lower bound of *one round* for both operations is *not* reached by the only other existing amnesic implementation [ACKM07] that albeit elegant, requires at least *three* rounds for reading and *two* for writing.

5.1.2 Contributions

Current state of the art leaves the following question open: *Do robust amnesic algorithms inherently have a non-optimal time complexity?* The thesis addresses this question and shows, for the first time, that amnesic algorithms can achieve optimal time complexity in both the $3t + 1$ and $4t + 1$ cases. Justified by the impossibility of robust and amnesic register constructions when readers do not write [CGK07], one of the key principles shared by our algorithms is having the readers change the base objects' state. The developed

algorithms are based on a novel concurrency detection mechanism and a helping procedure, by which a writer detects overlapping reads and helps them to complete. Specifically, we make the following two main contributions:

- A first algorithm, termed DMS, which uses $4t+1$ base objects, described in Section 5.3. With DMS, *every* (high-level) read and write operation is *fast*, i.e., it completes after only *one round* of communication with the base objects. This is the first robust and amnesic register construction (for unauthenticated data) with optimal time complexity.

- A second algorithm, termed DMS3, which uses the optimal number of $3t + 1$ base objects, presented in Section 5.4. With DMS3, every (high-level) read operation completes after only *two rounds*, while *write* operations complete after *three rounds*. This is the first amnesic and robust register construction (for unauthenticated data) with optimal read complexity. Note also that, compared to the optimal write complexity, it needs only one additional communication round.

Table 5.1 below summarizes our contributions and compares DMS and DMS3 with recent distributed storage solutions for unauthenticated data.

Table 5.1: Distributed storage for unauthenticated data

Protocol	Resilience	Worst-Case Time complexity Read	Write	Amnesic	Robust
[ACKM07]	$4t + 1$	3	2	√	√
DMS	$4t + 1$	1	1	√	√
[GV06]	$3t + 1$	2	2	×	√
[ACKM06]	$3t + 1$	$t + 1$	2	√	×
[GLV06]	$3t + 1$	unbounded	3	√	√
DMS3	$3t + 1$	2	3	√	√

5.2 Model and Preliminaries

5.2.1 Shared Memory Model

In this chapter we assume an asynchronous shared memory model consisting of a collection of processes interacting with a finite collection of n base objects. Up to t out of n base objects can suffer NR-arbitrary failures [JCT98] and any number of processes may fail by crashing. Each object implements

one or more *registers*. A register is an object type with value domains *Val*, an initial value v_0 and two invocations: *read*, whose response is $v \in Vals$ and *write*(v), $v \in Vals$, whose response is *ack*.

A read/write register is single-reader single-writer (SRSW) if only one process can read it and only one can write to it; a register is multi-reader single-writer (MRSW) if multiple processes can read it.

Sometimes processes need to perform two consecutive operations on the same base object, a write (of a register) followed by a read (of a different register). To minimize the number of rounds needed, we collapse consecutive write/read operations accessing the same base object to a single low-level operation called *write&read*. The *write&read* operation can be implemented in a single round, for instance relying on base objects with read-modify-write capabilities [CM05][1].

5.2.2 Preliminaries

In order to distinguish between the target register's interface and that of the base registers, throughout this chapter we denote the high-level read (resp. write) operation as READ (resp. WRITE). Each of the developed protocols uses an underlying layer that invokes operations on different base objects in separate threads in parallel. We use the notation from [ACKM06] and write **invoke** *write*(X_i,v) (resp. **invoke** $x[i] \leftarrow$ *read*(X_i)) to denote that a write(v) operation on register X_i (resp. a read of register X_i whose response will be stored in a local variable $x[i]$) is invoked in a separate thread by the underlying layer. The notation **invoke** $x[i] \leftarrow$ *write&read*$(\langle Y_i, v\rangle, X_i)$ denotes the invocation of an operation *write&read* on base object i, consisting of a write(v) on register Y_i followed by a read of register X_i (whose response will be stored in $x[i]$).

As base objects may be non-responsive, high-level operations can return while there are still pending invocations to the base objects. The underlying layer keeps track of which invocations are pending to ensure well-formedness, i.e., that a process does not invoke an operation on a base object while invocations of the same process and on the same base object are pending. Instead, the operation is denoted enabled. If an operation is enabled when a pending one responds, the response is discarded and the enabled operation is invoked. See e.g. [ACKM06] for a detailed implementation of such layers.

In order to better convey the insight behind the protocols, we simplify the presentation in two ways. We introduce a shared object termed *safe counter*

[1]Note that since *write&read* is not an atomic operation, it can be implemented from simple read/write registers.

and describe both algorithms in terms of this abstraction. Although easy to follow, the resulting implementations require more rounds than the optimal number. Thus, for each of the protocols we explain how with small changes these rather didactic versions can be "condensed" to achieve the announced time complexity. Secondly, for presentation simplicity we implement a SRSW register. Conceptually, a MRSW register for m readers can be constructed using m copies of this register, one for each reader. In a distributed storage setting, the writer accesses all m copies in parallel, whereas the reader accesses a single copy. It is worth noting that this approach is heavy and that in practice, cheaper solutions are needed to reduce the communication complexity and the amount of memory needed in the base objects.

We now introduce the *safe counter* abstraction used in our algorithms. A safe counter has two *wait-free operations* INC and GET. INC modifies the counter by incrementing its value (initially 0) and returns the new value. Specifically, the k^{th} INC operation denoted INC^k returns k. GET returns the current value of the counter without modifying it.

The counter provides the following guarantees:

Validity: If GET returns k then GET does not precede INC^k.

Safety: If INC^k precedes GET and for all $l > k$ GET precedes INC^l, then GET returns k.

Note that under concurrency, a safe counter might return an outdated value, but never a forged value. In the absence of concurrency, the newest value is returned.

We now explain the intuition behind our algorithms. Both algorithms use the safe counter introduced above to arbitrate between writer and reader. During each READ (resp. WRITE) operation, the reader (resp. writer) executes INC to advance the counter (resp. GET to read the counter). The values returned by the counter's operations are termed *views*. By incrementing its current view, a READ announces its intent to read from the base objects. A subsequent invocation of GET by the writer returns the updated view. When the writer detects a concurrent READ, indicated by a view change, it *freezes* the most recent value previously written. Freezing a value v means that v may be overwritten *only if* the READ operation that attempts to read v has completed.

We note that the READ operation that caused a value v to be frozen does not violate regularity by returning v because all newer values were written concurrently with the READ. However, READs must not return old values previously frozen. This is necessary to ensure regularity and it is done by

freezing a value v together with the view of the READ due to which v is frozen. A READ whose view is higher than the one associated with v knows that it must pick a newer value. A READ operation completes when it finds a value v to return such that (a) v is reported by a correct base object and (b) v is not older than the latest value written before the READ is invoked.

5.3 Fast Robust and Amnesic Storage

We start by describing an initial version of protocol DMS that uses the safe counter abstraction. It is worth noting that the algorithm requires more rounds than the optimum, but it conveys the main idea. Next, we explain the changes applied to DMS to obtain an algorithm with optimal time complexity.

5.3.1 Protocol Description

We present a robust and amnesic SRSW register construction using a safe counter and $4t+1$ regular base registers, out of which t can incur NR-arbitrary failures. Figure 5.1 illustrates a simple construction of the safe counter used. The description of the counter is omitted for the sake of brevity. The shared objects used by DMS are detailed in Figure 5.2 and the algorithm appears in Figure 5.3.

Predicates:
$\mathsf{safe}(c) \triangleq$
$|\{i : c' \in y[i] \wedge c' \geqslant c\}| \geqslant t+1$

GET()
for $1 \leqslant i \leqslant n$ **do** $y[i] \leftarrow \perp$
for $1 \leqslant i \leqslant n$ **do**
invoke $y[i] \leftarrow read(Y_i)$
wait for $n-t$ responses
return $\max\{c \in Integers : \mathsf{safe}(c)\}$

Local variables:
$y[1 \ldots n] \in Integers$
$k \in Integers$, initially 0

INC()
$k \leftarrow k+1$
for $1 \leqslant i \leqslant n$ **do**
invoke $write(Y_i, k)$
wait for $n-t$ responses
return k

Figure 5.1: Safe counter from $4t+1$ safe registers $Y_i \in Integers$.

The WRITE performs in two phases, (1) a write phase where it first writes a timestamp-value pair to $n-t$ registers and (2) a subsequent read phase, where it executes GET to read the current view. In case a view change occurs between two successive WRITEs, the value of the first WRITE is frozen. Recall that once frozen, a value is not erased before the next view change. Similarly, the READ consists of (1) a write phase, where it first executes INC to increment

the current view and (2) a subsequent read phase, where it reads at least $n-t$ registers. To ensure that READ never returns a corrupted value, the returned value must be read from $t+1$ registers, a condition captured by the predicate safe. Moreover, to ensure regularity, READ must not return old values written before the last WRITE preceding the READ. This condition is captured by the predicate highestCand.

We now give a more detailed description of the algorithm. As depicted in Figure 5.2, each base register consists of three value fields *current*, *prev* and *frozen* holding timestamp-value pairs, and an integer field *view*. The writer holds a variable x of the same type and uses x to overwrite the base registers. Each WRITE operation saves the timestamp-value pair previously written in $x.prev$. Then, it chooses an increasing timestamp, stores the value together with the timestamp in $x.curr$ and overwrites $n-t$ registers with x. Subsequently, the writer executes GET. If the view returned by GET is higher than the current view (indicating a concurrent READ), then $x.view$ is updated and the most recent value previously written is frozen, i.e., the content of $x.prev$ is stored in $x.frozen$ (line 14, Figure 5.3). Finally, WRITE returns *ack* and completes. It is important to note that the algorithm is amnesic because each correct base object stores at most three values (*curr*, *prev* and *frozen*).

The READ first executes INC to increment the current view, and then it reads at least $n-t$ registers into the array $x[1...n]$, where element i stores the content of register X_i. If necessary, it waits for additional responses until there is a *candidate* for returning, i.e., a read timestamp-value pair that satisfies both predicates safe and highestCand. A timestamp-value pair c is safe when it appears in some field *curr*, *prev* or *frozen* of $t+1$ elements of x, ensuring that c was reported by at least one correct register. Enforcing regularity is more subtle. Simply waiting until the highest timestamped value read becomes safe might violate liveness because it may be reported by a faulty register. To solve this problem, we introduce the predicate highestCand. A value c is highestCand when $2t+1$ base registers report values that were written *not after* c, which implies that newer values are missing from $t+1$ correct registers. As any complete WRITE skips at most t correct registers, all values newer than c were written *not* before READ is invoked and consequently, they can be discarded from the set of possible return candidates.

We now explain with help of Figure 5.4 why READs are wait-free. We consider the critical situation when multiple WRITEs are concurrent with a READ. Specifically, we consider the k^{th} READ (henceforth READ^k), whose INC results in k (henceforth INC^k), and the *last* WRITE that still reads a view lower than k, i.e., the corresponding GET returns a view lower than k. Note that by the safety property of the counter, INC^k does not precede

Types:
TSVals $\triangleq$ *Integers* $\times$ *Vals*, with selectors *ts* and *val*

Shared objects:
- regular registers $X_i \in TSVals^3 \times Integers$ with selectors *curr*, *prev*, *frozen* and *view*, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$
- safe counter object $Y \in Integers$, initially $Y = 0$

Figure 5.2: Shared objects used by DMS.

GET and thus c is stored in $2t + 1$ correct registers *before* any of them is read. A key aspect of the algorithm is to ensure that no matter how many WRITEs are subsequently invoked, c never disappears from all fields of those $2t + 1$ correct registers, as long as READ^k is still in progress. Essentially this holds because the subsequent WRITE re-writes c to all registers and it also freezes c to ensure that future WRITEs do the same. In this process, c migrates from *curr* to *prev* and from *prev* to *frozen* where it stays until the next view change. Therefore, c eventually becomes safe. But what if c is not highestCand? In this situation, at least $t + 1$ correct registers report timestamp-value pairs higher than c. We note that if any of them had stored c in its *frozen* field, then it would report c. This implies that none of these registers has stored c in its *frozen* field and thus, also none of these registers has stored a timestamp-value pair higher than c_h in its *curr* field. Therefore, c_h is reported by $t + 1$ correct registers, and hence it is safe. Note that c_h is also highestCand because only faulty registers report values with higher timestamps.

We now explain how the fast algorithm is derived from DMS. The principle underlying the optimization is to condense one round of write to the base objects and a subsequent round of read of the base objects into a single round of *write&read*. For this purpose we disregard the safe counter abstraction and directly weave INC and GET (Fig. 5.1) into READ and WRITE (Fig. 5.3) respectively. As a result, the reader advances the view *and* reads the base registers in one round. Likewise, the writer stores a value in the base registers *and* reads the view in a single round. The reader code (Fig. 5.3) is modified as follows: variable *view* is incremented locally, and line 3 is replaced with the statement **for** $1 \leqslant i \leqslant n$ **do invoke** $x[i] \leftarrow$ *write&read*$(\langle Y_i, view\rangle, X_i)$. Similarly, in the writer code (Fig. 5.3), line 9 is replaced with the statement **for** $1 \leqslant i \leqslant n$ **do invoke** $y[i] \leftarrow$ *write&read*$(\langle X_i, x\rangle, Y_i)$. Additionally in line 11, instead of executing GET, the writer picks the $(t + 1)$th highest element of y.

Predicates (reader):
$\mathsf{readFrom}(c, i) \triangleq (c = x[i].curr \wedge x[i].view < view) \vee$
$(c = x[i].frozen \wedge x[i].view = view)$
$\mathsf{safe}(c) \triangleq |\{i : c \in \{x[i].curr, x[i].prev, x[i].frozen\}\}| \geqslant t + 1$
$\mathsf{highestCand}(c) \triangleq |\{i : \mathsf{readFrom}(c', i) \wedge c'.ts \leqslant c.ts\}| \geqslant 2t + 1$

Local variables (reader):
$view \in Integers$, initially 0
$x[1 \ldots n] \in TSVals^3 \times Integers$

READ()
1 **for** $1 \leqslant i \leqslant n$ **do** $x[i] \leftarrow \perp$
2 $view \leftarrow \text{INC}(Y)$
3 **for** $1 \leqslant i \leqslant n$ **do invoke** $x[i] \leftarrow read(X_i)$
4 **wait** until $n - t$ responded $\wedge$ $\exists c \in TSVals$: $\mathsf{safe}(c) \wedge$ $\mathsf{highestCand}(c)$
5 return $c.val$

Local variables (writer):
$newView$, $ts \in Integers$, initially 0
$x \in TSVals^3 \times Integers$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

WRITE(v)
6 $ts \leftarrow ts+1$
7 $x.prev \leftarrow x.curr$
8 $x.curr \leftarrow \langle ts, v\rangle$
9 **for** $1 \leqslant i \leqslant n$ **do invoke** $write(X_i, x)$
10 **wait** for $n - t$ responses
11 $newView \leftarrow \text{GET}(Y)$
12 **if** $newView > x.view$ **then**
13 $x.view \leftarrow newView$
14 $x.frozen \leftarrow x.prev$
15 return ack

Figure 5.3: Robust and amnesic storage algorithm DMS $(4t + 1)$

We now informally argue that the optimization is correctness preserving. As in the above example, we consider READ^k and the last WRITE that reads a view lower than k. Recall that the WRITE operation stores c in $2t + 1$ correct base objects and each of them responds with the current view it has stored. The writer then picks the $(t + 1)$th highest view reported. We argue that $t + 1$ correct base objects have stored c before any of them respond to READ^k. This would imply that c is safe. As the WRITE operation reads a

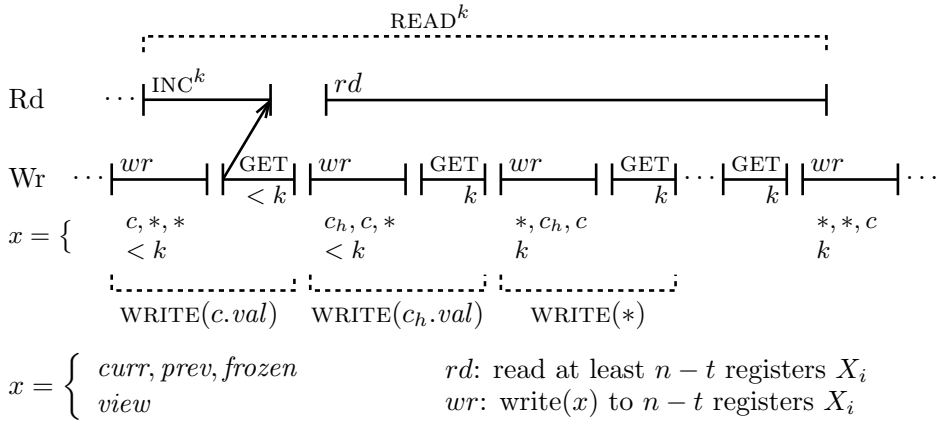


Figure 5.4: Correctness argument of the READ implementation in DMS

view lower than k, out of the $2t + 1$ correct base objects accessed by it, at most t report k. Thus, the remaining $t + 1$ objects are accessed by READ^k only after c was written to them. Applying the above arguments, it is not difficult to see that c is never erased from $t+1$ correct registers before READ^k completes, and thus it eventually becomes safe. Regarding regularity, again, arguments similar to above can be used. A formal proof of the optimized algorithm can be found in Appendix 5.5.

The remainder of this section is concerned with the correctness of DMS.

5.3.2 Protocol Correctness

Lemma 19 (Regularity)**.** *Algorithm DMS in Figure 5.3 implements a regular register.*

Proof. We show that the READ operation always returns the value of the latest WRITE preceding the READ, or a newer written value. Suppose that $c.val$ is the value returned by READ^k. We assume by contradiction that there exists a value $c_h.val$ such that $c_h.ts > c.ts$ and WRITE($c_h.val$) precedes READ^k. As WRITE($c_h.val$) is complete, $n-2t$ correct registers have stored c_h or a higher timestamp-value pair before any of them is read. The fact that $c.val$ is returned implies that c is highestCand. Thus, there are at least $2t+1$ registers X_i and values c' with timestamp $c'.ts \leqslant c.ts$ such that readFrom(c',i) is true. Note that one of them is a correct register X_i updated with c_h. As values are written with monotonically increasing timestamps, by definition of readFrom, necessarily c' is read from $x[i].frozen$ and $x[i].view = k$. However, because the counter is valid, the first time a WRITE operation reads view k is

only after the WRITE of $c_h.val$. Thus, in view k only timestamp-value pairs c_h or higher are frozen, a contradiction. □

Lemma 20 (Wait-freedom)**.** *Algorithm DMS in Figure 5.3 implements wait-free* READ *and* WRITE *operations.*

Proof. The WRITE operation is nonblocking because it never waits for more than $n - t$ responses. Showing that READs are also live is more subtle. To derive a contradiction, we assume that READ^k blocks at line 4 and show that there exists a candidate for returning. We consider the time after which all correct base objects (at least $3t + 1$) have responded. We choose c as the $2t + 1^{\text{st}}$ lowest timestamp-value pair readFrom a correct register. Note that c is highestCand by construction because values with timestamps $\leqslant c.ts$ are readFrom $2t + 1$ correct registers (set L). Also, we note that values with timestamps $\geqslant c.ts$ are readFrom $t + 1$ correct registers (set R). In the following, we distinguish the cases where the WRITE of $c.val$ reads a view equal to k (case 1), or lower than k (case 2). Note that by the validity of the counter, only views $\leqslant k$ are returned.

Case 1 implies that (a) only timestamp-value pairs lower than c are frozen, and (b) c is the highest timestamp-value pair readFrom the *curr* field of a correct register. Together (a) and (b) imply that c is the highest timestamp-value pair readFrom a correct register. Thus, for all registers $X_i \in R$ ($\geqslant t+1$), readFrom(c',i) implies that $c' = c$ and hence, c is safe. We now consider case 2 where WRITE($c.val$) reads a view lower than k. This implies that c or a higher timestamp-value pair is frozen in view k. If $t+1$ registers in L were updated with c before they are read, then they would report c either from their *curr* or their *frozen* field, and clearly c would be safe. Therefore, c is missing from $t + 1$ correct registers. Thus, WRITE($c.val$)'s write phase (lines 9–10) does not precede READ^k's read phase (lines 3–4). By the transitivity of the precedence relation, INC^k (line 2) precedes GET (line 11). By the safety of the counter, WRITE($c.val$) reads view k, a contradiction. □

Theorem 3 (Robustness)**.** *The algorithm in Figure 5.3 wait-free implements a regular register.*

Proof. Immediately follows from Lemma 19 and 20. □

5.4 An Optimally Resilient Algorithm

Similar to the previous section, we describe an initial version of DMS3 that uses a safe counter. The algorithm requires more rounds than the optimum but it is easier to understand because most of its complexity is hidden in

the counter implementation. Then, we overview the changes necessary to obtain the optimal algorithm. The full details of the optimized DMS3 such as the pseudocode and proofs can be found in Section 5.6. We proceed in a bottom-up fashion and describe the counter implementation first.

5.4.1 A Safe Counter with Optimal Resilience

We present a safe counter with operations INC and GET using $3t + 1$ base objects $i \in \{1 \dots n\}$, where t base objects can be subject to NR-arbitrary failures. The types and shared objects used by the counter are depicted in Figure 5.5 and the algorithm appears in Figure 5.6. Each base object i implements two regular registers: a register T_i holding a timestamp written by GET and read by INC, and a second register Y_i consisting of two fields pw and w, modified by INC and read by GET. While the pw field stores only the counter value, the w field stores the counter value together with a *high-resolution timestamp* [CGKV09]. A high-resolution timestamp is a timestamp-array with n entries, one for each base object.

Additional Types:

TSs $\triangleq$ *Integers* array of size n, *Integers*$[n]$

TSsInt $\triangleq$ *TSs* $\times$ *Integers* with selectors *hrts* (high-resolution timestamp) and *cnt*

Shared objects:

- regular registers $Y_i \in$ *Integers* $\times$ *TSsInt* with selectors pw and w, initially $Y_i = \langle 0, \langle [0, \dots, 0], 0 \rangle \rangle$
- regular registers $T_i \in$ *Integers*, initially 0

Figure 5.5: Shared objects used by the safe counter $(3t + 1)$

The GET operation performs in two phases. The first phase reads from the base objects until $n - t$ registers Y_i have responded and all responses are *non-conflicting*. This condition is captured by the predicate conflict. When two base objects i and j are in conflict, then at least one of them is malicious. In this situation, the GET operation can wait for more than $n - t$ responses without blocking, effectively filtering out responses from malicious base objects. Next, the GET operation uses the responses to build a candidate set from values appearing in the w field of Y_i. In the second phase, the GET operation chooses an increasing timestamp ts and overwrites $n-t$ registers T_i with ts; at the same time it re-reads the registers Y_i until $n - t$ of them have responded and there exists a *candidate* to return. This condition is captured

by the predicates safe and highCand. If no candidate can be returned (because of overlapping INC operations), GET returns the initial counter value 0.

Similarly, the INC operation performs in two phases, a pre-write and a write phase. The pre-write phase accesses $n - t$ base objects i, overwriting the pw field of Y_i with an increasing counter value and reading the individual timestamps stored in T_i into a single high-resolution timestamp. Subsequently, in the write phase, INC stores the counter value together with the high-resolution timestamp in the w field of $n - t$ registers Y_i and returns.

We now show that the algorithm in Figure 5.6 wait-free implements a safe counter. We do this by showing that the two following properties are satisfied:

Validity: If GET returns k then GET does not precede INC^k.

Safety: If INC^k precedes GET and for all $l > k$ GET precedes INC^l, then GET returns k.

Lemma 21 (Validity)**.** *The counter object implemented in Figure 5.6 is valid.*

Proof. If the initial value is returned then we are done. Else only a value $c.cnt = k$ is returned such that c is safe. This implies that $t + 1$ base objects report values k or higher either from their pw or w fields. As not all of them are faulty, there exists a correct object Y_i and a value $l \geqslant k$ such that l was indeed written to Y_i. As INC^k precedes INC^l (or it is the same operation) and GET does not precede INC^l, it follows that GET does not precede INC^k. □

Lemma 22 (Safety)**.** *The counter object implemented in Figure 5.6 is safe.*

Proof. Let INC^k be the last operation preceding the invocation of GET. Furthermore, for all $l > k$, GET precedes INC^l. By assumption, $c.cnt = k$ was written to the w field of $t+1$ correct objects before GET is invoked. Therefore, c is added to the candidate set C (line 16) and because at most $2t$ objects respond without c, it is never removed. Furthermore, $t + 1$ correct objects eventually report c in the second GET round and c becomes safe. As there are no concurrent INC operations, eventually $2t + 1$ correct objects report values k or lower from their w field and hence all c_h where $c_h.cnt > k$ are removed from C. Thus, c eventually becomes both safe and highCand and $c.cnt = k$ is returned. □

Lemma 23 (Wait-freedom)**.** *The counter object implemented in Figure 5.6 is wait-free.*

Local variables (INC):
$y \in \mathit{Integers} \times \mathit{TSsInt}$, initially $\langle 0, \langle [0, \ldots, 0], 0 \rangle \rangle$
$\mathit{cnt} \in \mathit{Integers}$, initially 0 //*counter value*
$\mathit{hrts}[1 \ldots n] \in \mathit{Integers}$, initially $[0, \ldots, 0]$ //*high-res timestamp*

INC()
1 $\mathit{cnt} \leftarrow \mathit{cnt} + 1$
2 $y.\mathit{pw} \leftarrow \mathit{cnt}$
3 **for** $1 \leqslant i \leqslant n$ **do invoke** $\mathit{hrts}[i] \leftarrow \mathit{write\&read}(\langle Y_i, y \rangle, T_i)$
4 **wait** for $n - t$ responses
5 $y.w.\mathit{hrts} \leftarrow \mathit{hrts}$
6 $y.w.\mathit{cnt} \leftarrow \mathit{cnt}$
7 **for** $1 \leqslant i \leqslant n$ **do invoke** $\mathit{write}(Y_i, y)$
8 **wait** for $n - t$ responses
9 return *ack*

Predicates (GET):
$\mathsf{conflict}(i, j) \triangleq y[i].w.\mathit{hrts}[j] \geqslant \mathit{ts}$
$\mathsf{safe}(c) \triangleq |\{i : \max\{PW[i]\} \geqslant c.\mathit{cnt} \vee (\exists c' \in W[i] \wedge c'.\mathit{cnt} \geqslant c.\mathit{cnt})\}| > t$
$\mathsf{highCand}(c) \triangleq c \in C \wedge (c.\mathit{cnt} = \max\{c'.\mathit{cnt} : c' \in C\})$

Local variables (GET):
$PW[1 \ldots n] \in 2^{\mathit{Integers}}$, $W[1 \ldots n] \in 2^{\mathit{TSsInt}}$, $C \in 2^{\mathit{TSsInt}}$
$y[1 \ldots n] \in \mathit{Integers} \times \mathit{TSsInt} \cup \{\perp\}$
$\mathit{ts} \in \mathit{Integers}$, initially 0

GET()
10 **for** $1 \leqslant i \leqslant n$ **do** $y[i] \leftarrow \perp$; $PW[i] \leftarrow W[i] \leftarrow \varnothing$
11 $C \leftarrow \varnothing$
12 $\mathit{ts} \leftarrow \mathit{ts} + 1$
13 **for** $1 \leqslant i \leqslant n$ **do invoke** $y[i] \leftarrow \mathit{read}(Y_i)$
repeat
14 CHECK
15 **until** a set S of $n - t$ objects responded $\wedge\ \forall i, j \in S : \neg\mathsf{conflict}(i, j)$
16 $C \leftarrow \{y[i].w : |\{j : y[j].w \neq y[i].w\}| \leqslant 2t\}$
17 **for** $1 \leqslant i \leqslant n$ **do invoke** $y[i] \leftarrow \mathit{write\&read}(\langle T_i, \mathit{ts} \rangle, Y_i)$
repeat
18 CHECK
19 $C \leftarrow C \backslash \{c \in C : |\{i : \exists c' \in W[i] \wedge c' \neq c\}| \geqslant 2t + 1\}$
20 **until** $n - t$ responded $\wedge\ \exists c \in C$: $(\mathsf{safe}(c) \wedge \mathsf{highCand}(c)) \vee C = \varnothing$
21 **if** $C \neq \varnothing$ **then** return $c.\mathit{cnt}$ **else** return 0

CHECK
if Y_i responded **then**
$PW[i] \leftarrow PW[i] \cup \{y[i].\mathit{pw}\}$
$W[i] \leftarrow W[i] \cup \{y[i].w\}$

Figure 5.6: Safe counter algorithm $(3t + 1)$

Proof. As the INC operation never waits for more than $n-t$ responses, clearly it never blocks. In the following we prove that the GET operation does not block (1) at line 15 and (2) at line 18. We assume by contradiction that the GET operation blocks. Case (1): as the GET operation never updates a correct base object with ts before the second round, correct base objects are never in conflict with each other and thus the GET operation does not block at line 15. Case (2): The GET operation blocks at line 18. Therefore, there exists $c \in C$ and c is not safe. Let $c.cnt = k$. If some correct base object has reported c in its w field in the first round of GET, then $t+1$ correct base objects report k or higher in their pw field in the second round and thus c is safe. Therefore, we assume that no correct base object reports c in w in the first round. If no correct object reports c in w in the second round, then $2t+1$ correct base objects respond with $c' \neq c$ in their w field and c is removed from C. In the following we assume that some correct object reports c in w in the second round. Let F ($|F| > 0$) denote the set of faulty objects that report c in their w field in the first round. Let X ($|X| \geqslant 0$) be the set of correct base objects i such that Y_i reports to the second GET round a value lower than k in both fields pw and w. This implies that the pre-write phase of INC at Y_i does not precede the second GET round reading Y_i (see Fig. 5.7 (a)). By the semantics of *write&read*, the second GET round has updated T_i with ts *before* reading Y_i (line 17). Similarly, the first round of INC has pre-written k to Y_i *before* reading T_i (line 3). By transitivity, the second GET round has completed the update of T_i *before* the first INC round has read T_i, and thus T_i reports ts (Fig. 5.7 (a)). Let $X' = \{j \in X : c.hrts[j] = ts\}$, that is, the objects in X that have actually responded to the first INC round. Note that for all $i \in F$ and for all $j \in X'$, conflict(i, j) is true. Hence, the $2t+1-|F|$ objects that have responded without c in their w field in the first round of GET do not include any object in X'. Overall, after the second GET round, $2t+1-|F|+|X'|$ base objects have responded without c in their w field. If $|F| \leqslant |X'|$ then c is removed from the set of candidates C (line 20), a contradiction. Therefore, we consider the case $|F| > |X'|$. Out of the $t+1$ correct base objects updated by the pre-write phase of INC, $t+1-|X'|$ respond with a timestamp lower than ts. Consequently, for every such base object i, GET has completed updating T_i with ts *not before* INC reads T_i (see Figure 5.7 (b)). By the semantics of *write&read* and by the transitivity of the precedence relation, register Y_i has stored k in its pw field before the second GET round reads Y_i. Hence, at least $t+1-|X'|+|F|$ base objects report values k or higher. As $|F| > |X'|$, $t+1$ base objects report k or a higher value, and thus c is safe, a contradiction.

□

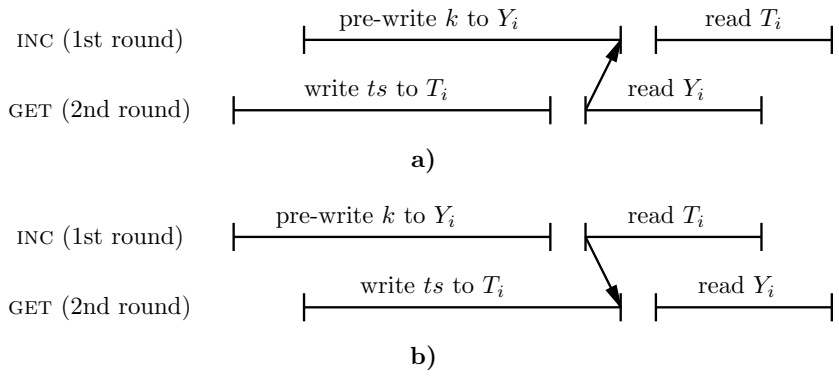


Figure 5.7: Safe counter correctness argument

Theorem 4. *The Algorithm in Figure 5.6 wait-free implements a safe counter.*

Proof. Follows directly from Lemma 21, 22 and 23. □

5.4.2 The DMS3 Protocol

Protocol Description

In this section we present a robust and amnesic SRSW register construction from a safe counter and $3t + 1$ regular base registers, out of which t can be subject to NR-arbitrary failures. We now describe the WRITE and READ operations of the DMS3 algorithm illustrated in Figure 5.8.

The WRITE operation performs in three phases, (1) a pre-write phase (lines 7–9) where it stores a timestamp-value pair c in the pw field of $n - t$ registers, (2) a read phase (line 10), where it calls GET to read the current view and (3) a write phase (lines 14–16), where it overwrites the w field of $n - t$ registers with c. If the read phase results in a view change, the most recent value previously written is frozen together with the new view. This is done by updating the *view* field and copying the value stored in w to the *frozen* field (lines 11–13). The reader performs exactly the same steps as in DMS (see Section 5.3).

We now explain with help of Figure 5.9 why READs are wait free. Similar to the description of DMS in Section 5.3, we consider READ^k and the last WRITE that reads a view lower than k. Note that INC^k does not precede GET and thus, c is stored in the pw field of $t + 1$ correct registers before they are read. Also, the w field of $t + 1$ correct registers is updated with c. As the subsequent WRITE encounters a view change, c is written to the *frozen* field

Shared objects:
regular registers $X_i \in TSVals^3 \times Integers$, with selectors *pw*, *w*, *frozen* and *view*, initially $X_i = \langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

Predicates (reader):
$\mathsf{readFrom}(c, i) \triangleq (c = x[i].w \wedge x[i].view < view) \vee (c = x[i].frozen \wedge x[i].view = view)$
$\mathsf{safe}(c) \triangleq |\{i : c \in \{x[i].pw, x[i].w, x[i].frozen\}\}| \geqslant t + 1$
$\mathsf{highestCand}(c) \triangleq |\{i : \mathsf{readFrom}(c', i) \wedge c'.ts \leqslant c.ts\}| \geqslant 2t + 1$

Local variables (reader):
$view \in Integers$, initially 0
$x[1 \ldots n] \in TSVals^3 \times Integers \cup \{\perp\}$

READ()
1 **for** $1 \leqslant i \leqslant n$ **do** $x[i] \leftarrow \perp$
2 $view \leftarrow \text{INC}(Y)$
3 **for** $1 \leqslant i \leqslant n$ **do invoke** $x[i] \leftarrow read(X_i)$
4 **wait** until $n - t$ responded $\wedge\ \exists c \in TSVals$: $\mathsf{safe}(c) \wedge \mathsf{highestCand}(c)$
5 return $c.val$

Local variables (writer):
ts, $newView \in Integers$, initially 0
$x \in TSVals^3 \times Integers$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

WRITE(v)
6 $ts \leftarrow ts+1$
7 $x.pw \leftarrow \langle ts, v\rangle$
8 **for** $1 \leqslant i \leqslant n$ **do invoke** $write(X_i, x)$
9 **wait** for $n - t$ responses
10 $newView \leftarrow \text{GET}(Y)$
11 **if** $newView > x.view$ **then**
12 $x.view \leftarrow newView$
13 $x.frozen \leftarrow x.w$
14 $x.w \leftarrow \langle ts, v\rangle$
15 **for** $1 \leqslant i \leqslant n$ **do invoke** $write(X_i, x)$
16 **wait** for $n - t$ responses
17 return *ack*

Figure 5.8: Robust and amnesic storage algorithm DMS3 ($3t+1$)

of $t + 1$ correct registers, where it stays until READ^k completes. Hence, c is sampled from $t + 1$ correct registers' *pw, w* or *frozen* field and thus it is safe. Note that c is also highestCand because only faulty registers report newer

values.

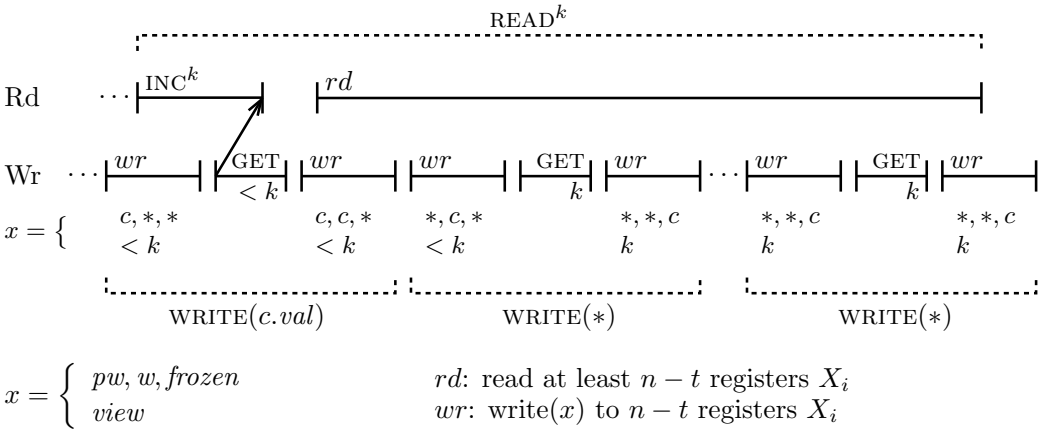


Figure 5.9: Correctness argument of the READ implementation in DMS3

With DMS3, the high-level operations have a non-optimal time complexity. We now explain how the optimized version is obtained by collapsing individual low-level operations. More precisely, a write operation and a consecutive read operation are merged together to a *write&read* operation. The safe counter abstraction is disregarded and the counter operations INC and GET are weaved into READ and WRITE respectively. Recall that the counter operations consist of two rounds each. In the WRITE implementation, the pre-write phase and the first round of GET are collapsed. Note that the three-phase structure of the WRITE is preserved in that the writer reads the current view *before* it moves to the write phase. Similarly, in the READ implementation, the second INC round and the read phase are merged together. Overall, this results in a time complexity of *three* rounds for the WRITE and *two* rounds for the READ.

We now informally argue that the optimization is correctness preserving. As above, we consider READ^k and the last WRITE that reads a view lower than k. We argue that $t + 1$ correct base registers have stored c in their pw field before any of them is read. This would imply that c is safe. The fact that the WRITE of $c.val$ reads a view lower than k implies that k is missing from at least $2t + 1$ base objects. We know from the safe counter algorithm in the previous section that if only $2t$ base objects respond without k, then k is never removed from the set of candidates. As the safe counter implementation is wait-free, k is eventually read, contradicting the initial assumption. Therefore, $2t + 1$ base objects respond without k, and thus there are $t + 1$ correct base objects among them that are accessed by (the

read phase of) READ^k only after c was pre-written to them. By applying similar arguments as above, it is not difficult to see that c does not disappear from any of the $t+1$ correct base objects before READ^k completes. This would imply that c eventually becomes safe. For a formal treatment we refer the interested reader to Appendix 5.6. The remainder of this section is concerned with the correctness of DMS3.

Protocol Correctness

Lemma 24 (Regularity)**.** *Algorithm DMS3 in Figure 5.8 implements a regular register.*

Proof. Identical to the proof of Lemma 19. □

Lemma 25 (Wait-freedom)**.** *Algorithm DMS3 in Figure 5.8 implements wait-free* READ *and* WRITE *operations.*

Proof. The WRITE operation is nonblocking because it never waits for more than $n-t$ responses. To derive a contradiction we assume that READ^k blocks at line 4 and show that there exists a candidate for returning. We consider the time after which all correct base objects (at least $2t+1$) have responded. We choose c as the highest timestamp-value pair readFrom a correct register. Note that c is highestCand by construction because values with timestamps $\leqslant c.ts$ are readFrom $2t+1$ correct registers. In the following, we distinguish the cases where the view read by the WRITE of $c.val$ is equal to k (case 1) or it is lower than k (case 2). Note that by the validity of the counter, only views $\leqslant k$ are returned. Case 1: Let X_i be a correct register such that readFrom(c, i). Since by assumption $x[i].view = k$, c is readFrom the *frozen* field of X_i. However, in view k only timestamp-value pairs lower than c are frozen, a contradiction. Now we consider case 2, where the WRITE($c.val$) reads a view lower than k. This implies that INC^k does not precede GET. As the pre-write phase (lines 8–9) precedes GET (line 10), and INC^k (line 2) precedes the read phase (lines 3–4), by transitivity, the pre-write phase also precedes the read phase (see Figure 5.9). Thus, $t+1$ correct registers have stored c in their pw field *before* they are read. What is left to show is that no subsequent WRITE erases c from all fields of those $t+1$ correct registers. Note that in view k, only timestamp-value pairs c or higher a frozen. Thus, if c was stored in the w field of $t+1$ correct registers before they are read, then c would be safe. Hence, c is missing from $t+1$ correct registers' w field. Consequently, WRITE($c.val$)'s write phase (lines 15–16) does not precede READ^k's read phase (lines 3–4). By transitivity, the subsequent WRITE reads view k and freezes c. Note that c is erased from pw only after c was previously stored in w

(line 14). Furthermore, c is erased from w only after it was stored in *frozen* (line 13). As k is the last view, by the validity of the safe counter, c is never erased from *frozen*.

□

Theorem 5 (Robustness)**.** *Algorithm DMS3 in Figure 5.8 implements a robust register.*

Proof. Immediately follows from Lemma 24 and 25. □

5.5 The Optimized DMS Protocol

In this section we prove the optimized algorithm DMS that achieves a worst-case latency of *one* round for read/write access to Byzantine storage.

Lemma 26 (Regularity)**.** *The algorithm in Figure 5.10 implements a regular register.*

Proof. We show that the READ operation never returns a value older than the latest value written before the READ is invoked. Suppose that $c.val$ is the value returned by READ^k. We assume by contradiction that there exists a value $c_h.val$ such that $c_h.ts > c.ts$ and WRITE($c_h.val$) precedes READ^k. Hence, $n - 2t$ correct registers have stored c_h or a higher timestamp-value pair before any of them is read. The fact that $c.val$ is returned implies that c is highestCand. Thus, there are at least $2t + 1$ registers X_i and values c' with timestamp $c'.ts \leqslant c.ts$ such that readFrom(c',i) is true. Note that one of them is a correct register X_i updated with c_h. As values are written with monotonically increasing timestamps, by definition of readFrom, necessarily c' is read from $x[i].frozen$ and $x[i].view = k$. As by assumption WRITE of $c_h.val$ precedes READ^k, safeView is false for any view $\geqslant k$. Thus, the first time a WRITE operation reads view k is only after the WRITE($c_h.val$). Hence, in view k only timestamp-value pairs c_h or higher are frozen, a contradiction.

□

Lemma 27 (Wait-freedom)**.** *The algorithm in Figure 5.10 implements wait-free* READ *and* WRITE *operations.*

Proof. The WRITE operation is nonblocking because it never waits for more than $n - t$ responses. Showing that READs are also live is more involved. To derive a contradiction, we assume that READ^k blocks at line 5 and show that there exists a candidate for returning. We consider the time after which all correct base objects (at least $3t + 1$) have responded. We choose c as

Predicates (reader):
$\mathsf{readFrom}(c, i) \triangleq (c = x[i].curr \wedge x[i].view < view) \ \vee \ (c = x[i].frozen \wedge x[i].view = view)$
$\mathsf{safe}(c) \triangleq |\{i : c \in \{x[i].curr, x[i].prev, x[i].frozen\}\}| \geqslant t + 1$
$\mathsf{highestCand}(c) \triangleq |\{i : \mathsf{readFrom}(c', i) \wedge c'.ts \leqslant c.ts\}| \geqslant 2t + 1$

Local variables (reader):
$view \in Integers$, initially 0
$x[1 \ldots n] \in TSVals3 \times Integers$

READ()
1 **for** $1 \leqslant i \leqslant n$ **do** $x[i] \leftarrow \perp$
2 $view \leftarrow view + 1$
3 **for** $1 \leqslant i \leqslant n$ **do**
4 **invoke** $x[i] \leftarrow write\&read(\langle Y_i, view\rangle, X_i)$
5 **wait** until $n - t$ responded $\wedge \ \exists c \in TSVals$: $\mathsf{safe}(c) \wedge \mathsf{highestCand}(c)$
6 return $c.val$

Predicates (writer):
$\mathsf{safeView}(c) \triangleq |\{i : c' \in y[i] \wedge c' \geqslant c\}| > t$

Local variables (writer):
$newView$, $ts \in Integers$, initially 0
$y[1 \ldots n] \in Integers \cup \{\perp\}$
$x \in TSVals^3 \times Integers$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

WRITE(v):
7 **for** $1 \leqslant i \leqslant n$ **do** $y[i] \leftarrow \perp$
8 $ts \leftarrow ts+1$
9 $x.prev \leftarrow x.curr$
10 $x.curr \leftarrow \langle ts, v\rangle$
11 **for** $1 \leqslant i \leqslant n$ **do**
12 **invoke** $y[i] \leftarrow write\&read(\langle X_i, x\rangle, Y_i)$
13 **wait** for $n - t$ responses
14 $newView \leftarrow \max\{c \in Integers$: $\mathsf{safeView}(c)\}$
15 **if** $newView > x.view.curr$ **then**
16 $x.view \leftarrow newView$
17 $x.frozen \leftarrow x.prev$
18 return ack

Figure 5.10: The optimized DMS protocol $(4t + 1)$

the $(2t + 1)$th lowest timestamp-value pair readFrom a correct register. Note that c is highestCand by construction because values with timestamps $\leqslant c.ts$

are readFrom $2t + 1$ correct registers (set L). Also, we note that values with timestamps $\geqslant c.ts$ are readFrom $t + 1$ correct registers (set R). In the following, we distinguish the cases when the WRITE of $c.val$ reads a view equal to k (case 1), or lower than k (case 2). As READ^k is blocking, only views $\leqslant k$ are returned. Case 1 implies that (a) only timestamp-value pairs lower than c are frozen, and (b) c is the highest timestamp-value pair readFrom the *curr* field of a correct register. Together (a) and (b) imply that c is the highest timestamp-value pair readFrom a correct register. Thus, for all registers $X_i \in R$ ($\geqslant t + 1$), readFrom(c',i) implies that $c' = c$ and hence, c is safe.

We now consider case 2, where WRITE($c.val$) reads a view $< k$. If no value is written after $c.val$, then all registers in R report c and thus c is safe. Let $c_h > c$ be the immediate successor of c. By the choice of c, c_h is missing from the fields of all registers in L (at least $2t + 1$). Therefore, there exist $t + 1$ base objects $i \in L$ such that the write of c_h to register X_i (line 12) *does not* precede the read of X_i (line 4). By the semantics of *write&read*, the write of view k to Y_i precedes the read of X_i (line 4) and the write of c_h to X_i precedes the read of Y_i (line 12) at any of those $t + 1$ objects. By transitivity of the precedence relation, Y_i has stored view k before it is read (see Fig. 5.11). Thus, the WRITE of $c_h.val$ reads view k from a subset of $t+1$ correct objects. Therefore, safeView(k) holds and hence, c_h reads view k. We note that no value higher that c_h is readFrom any correct register. Thus, for all values c' readFrom a register in R (at least $t + 1$), $c' \in \{c, c_h\}$. Moreover, if $c' = c_h$, then $c = x[i].prev$ and thus c is safe.

□

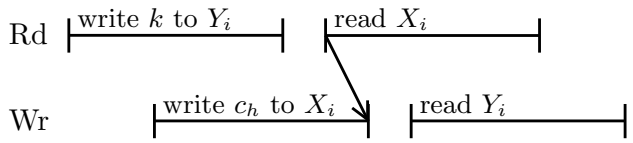


Figure 5.11: Real-time ordering of read/write operations on base object i

Theorem 6 (Robustness)**.** *The algorithm in Figure 5.10 wait-free implements a regular register.*

Proof. Immediately follows from Lemma 26 and 27. □

5.6 The Optimized DMS3 Protocol ($3t + 1$)

In this section we prove the optimized protocol DMS3 that achieves the optimal worst-case complexity of *two* rounds for every read operation.

Predicates:
$\mathsf{readFrom}(c, i) \triangleq (c = x[i].w \wedge x[i].view < view)$
$\vee\ (c = x[i].frozen \wedge x[i].view = view)$
$\mathsf{safe}(c) \triangleq |\{i : c \in \{x[i].pw, x[i].w, x[i].frozen\}\}| \geqslant t + 1$
$\mathsf{highestCand}(c) \triangleq |\{i : \mathsf{readFrom}(c', i) \wedge c'.ts \leqslant c.ts\}| \geqslant 2t + 1$

Local variables:
$y \in Integers \times TSsInt$, initially $\langle 0, \langle [0, \ldots, 0], 0\rangle\rangle$
$view \in Integers$, initially 0
$x[1 \ldots n] \in TSVals^3 \times Integers \cup \{\perp\}$
$hrts[1 \ldots n] \in Integers$ //*high-resolution-timestamp*

READ()
1 **for** $1 \leqslant i \leqslant n$ **do** $x[i] \leftarrow \perp$
2 $view \leftarrow view + 1$
3 $y.pw \leftarrow view$
4 **for** $1 \leqslant i \leqslant n$ **do invoke** $hrts[i] \leftarrow write\&read(\langle Y_i, y\rangle, T_i)$
5 **wait** for $n - t$ responses
6 $y.w.hrts \leftarrow hrts$
7 $y.w.cval \leftarrow view$
8 **for** $1 \leqslant i \leqslant n$ **do invoke** $x[i] \leftarrow write\&read(\langle Y_i, y\rangle, X_i)$
9 **wait** until $n - t$ responded $\wedge\ \exists c \in TSVals$: $\mathsf{safe}(c) \wedge \mathsf{highestCand}(c)$

Figure 5.12: Optimized READ operation

Lemma 28 (Regularity)**.** *The algorithm in Figures 5.12 and 5.13 implements a regular register.*

Proof. Identical to the proof of Lemma 26. □

Lemma 29 (Wait-freedom 1)**.** *The algorithm in Figures 5.12 and 5.13 implements wait-free* WRITE *operations.*

Proof. A simple extension of Lemma 23 proves that the algorithm wait-free implements the WRITE operation. □

Lemma 30 (Wait-freedom 2)**.** *The algorithm in Figures 5.12 and 5.13 implements wait-free* READ *operations.*

Predicates (get):
$\mathsf{conflict}(i,j) \triangleq y[i].w.hrts[j] \geqslant ts$
$\mathsf{safeView}(c) \triangleq |\{i : \max\{PW[i]\} \geqslant c.cval \vee (\exists c' \in W[i] \wedge c'.cval \geqslant c.val)\}| > t$
$\mathsf{highCandView}(c) \triangleq c \in C \wedge (c.cval = \max\{c'.cval : c' \in C\})$

Local variables:
$PW[1 \ldots n] \in 2^{Integers}$, $W[1 \ldots n] \in 2^{TSsInt}$, $C \in 2^{TSsInt}$
$y[1 \ldots n] \in Integers \times TSsInt \cup \{\perp\}$
ts, $newView \in Integers$, initially 0
$x \in TSVals^3 \times Integers$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

WRITE(v):
1 **for** $1 \leqslant i \leqslant n$ **do** $y[i] \leftarrow \perp$; $PW[i] \leftarrow W[i] \leftarrow \varnothing$
2 $C \leftarrow \varnothing$
3 $ts \leftarrow ts{+}1$
4 $x.pw \leftarrow \langle ts, v\rangle$
5 **for** $1 \leqslant i \leqslant n$ **do invoke** $y[i] \leftarrow write\&read(\langle X_i, x\rangle, Y_i)$
6 **repeat** CHECK
7 **until** a set S of $n-t$ objects responded $\wedge\ \forall i,j \in S : \neg\mathsf{conflict}(i,j)$
8 $C \leftarrow \{y[i].w : |\{j : y[j].w \neq y[i].w\}| \leqslant 2t\}$
9 **for** $1 \leqslant i \leqslant n$ **do invoke** $y[i] \leftarrow write\&read(\langle T_i, ts\rangle, Y_i)$
10 **repeat**
11 CHECK
12 $C \leftarrow C \backslash \{c \in C : |\{i : \exists c' \in W[i] \wedge c' \neq c\}| \geqslant 2t+1\}$
13 **until** $n-t$ responded $\wedge\ \exists c \in C$: ($\mathsf{safeView}(c) \wedge \mathsf{highCandView}(c)$) $\vee\ C = \varnothing$
14 **if** $C \neq \varnothing$ **then** $newView \leftarrow c.cval$
15 **if** $newView > x.view$ **then**
16 $x.view \leftarrow newView$
17 $x.frozen \leftarrow x.w$
18 $x.w \leftarrow \langle ts, v\rangle$
19 **for** $1 \leqslant i \leqslant n$ **do invoke** $write(X_i, x)$
20 **wait** for $n-t$ responses
21 return ack

CHECK
if Y_i responded **then**
$PW[i] \leftarrow PW[i] \cup \{y[i].pw\}$
$W[i] \leftarrow W[i] \cup \{y[i].w\}$

Figure 5.13: Optimized WRITE operation

Proof. As the first round of READ never waits for more than $n-t$ responses it never blocks at line 5 in Figure 5.12. We assume by contradiction that some READ with view k blocks at line 9 in Figure 5.12. We show that there exists a timestamp-value pair c such that c is both highestCand and safe. Let $c.val$ be the last value written in a view lower than k. If no value was written in a view lower than k, then let $c.val$ be the initial value. Note that c exists because the fields of every correct object are initialized to $\langle 0, v_0 \rangle$ in view 0 and $k > 0$. Moreover, as k is the last view, there are only finitely many WRITE operations such that safeView does not hold for k. We first show that c is safe. If $c.val$ is not the initial value then $c.val$ was written in a view lower than k. We partition the set of correct objects read by the reader (at least $2t+1$) into two sets O and $\overline{O}$ where O consists of all correct objects i such that the pre-write of c to X_i (line 5, Fig. 5.13) precedes the read of X_i (line 8, Fig. 5.12) and $\overline{O}$ contains all other correct objects. In the following we consider the two possible cases (i) $|O| < t+1$ and (ii) $|O| \geqslant t+1$. If $c.val$ is the initial value then only case (ii) needs to be considered.

Case (i): $|O| < t+1$ implies that $|\overline{O}| \geqslant t+1$ and for all $i \in \overline{O}$ holds that the pre-write of c to X_i (line 5, Fig. 5.13) *does not* precede the read of X_i (line 8, Fig. 5.12). By the semantics of *write&read*, the write of view k to Y_i precedes the read of X_i (line 8, Fig. 5.12) and the pre-write of c to X_i precedes the first round read of Y_i (line 5, Fig. 5.13) at all objects $i \in \overline{O}$. By transitivity of the precedence relation, Y_i has stored view k in its w field before it is read (see Fig. 5.14). Thus, the WRITE of $c.val$ reads view k from field $y[j].w$ of some correct object $j \in \overline{O}$. Therefore k is added to candidate set C (line 8, Fig. 5.13). As k is the last view, $t+1$ correct objects report k and all higher candidate views are removed from C. Thus, k is both safeView and highCandView. Hence, $c.val$ is written in view k, a contradiction.

Case (ii): For every $i \in O$, c has been stored in register X_i *before* X_i was read. If $c.val$ is the last value written, then c can be read from the pw field of X_i and thus c is safe. Note that the same holds for the initial value. What is left to show is that no subsequent WRITE operation erases c from all fields of variable x. Note that c is erased from $x.pw$ only after c was previously stored in $x.w$. (line 18, Fig. 5.13). Furthermore, as the subsequent WRITE reads view k, c is erased from w only after it is stored in $x.frozen$ (line 17, Fig. 5.13). As k is the last view, safeView never holds for a view $> k$ and thus c is never erased from $x.frozen$. We now show that c is also highestCand. By the choice of c and by definition of readFrom, only timestamp-value pairs $c'.ts \leqslant c.ts$ are readFrom a correct register's w or *frozen* fields. As there exist $2t+1$ correct registers, c is highestCand. □

Theorem 7 (Robustness)**.** *The algorithm in Figures 5.12 and 5.13 wait-free*

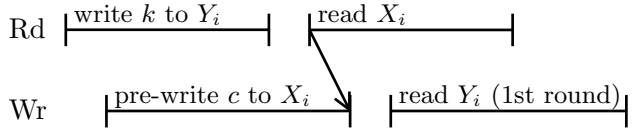


Figure 5.14: Real-time ordering of read/write operations on base object i

implements a regular register.

Proof. Immediately follows from Lemma 28, 29 and 30. □

5.7 Summary of the Results

We have presented amnesic algorithms that robustly implement a shared register from a collection of n base objects, of which up to $t < n/3$ can be subject to NR-arbitrary failures. For $n \geqslant 3t + 1$ we have shown that *two* rounds of communication with the base objects are sufficient for every READ operation to complete. This is the first robust and amnesic register construction that matches the *two*-round lower bound proved in [GV06]. For the $n \geqslant 4t + 1$ case, we have presented the first robust and amnesic register construction that matches the (trivial) *one*-round lower bound for *every* operation. We note that our construction is tight because with less than $4t + 1$ base objects, both the READ and the WRITE operations require at least *two* communication rounds [ACKM06, GV06].

The main result, namely that robust access to amnesic storage is possible in optimal time is somewhat surprising given the large body of literature on non-amnesic or non-robust algorithms. Moreover, our result is counter-intuitive: having each read also modify the based objects does not prevent fast read implementations (supporting any number of readers). As a corollary, our result suggests that the intuition of amnesic algorithms being inherently less efficient than non-amnesic ones is largely unjustified.

Chapter 6

Robust Storage with Secret Tokens

In this chapter we present robust algorithms that reduce the time complexity and improve the scalability of unauthenticated storage.

Our algorithms make use of secret tokens, which are values randomly selected by the clients and attached to the data written. Tokens are secret because they cannot be predicted by the adversary before they are used, and thus revealed, by the clients. The developed algorithms do not rely on unproven cryptographic assumptions as algorithms based on self-verifying data. They are optimally-resilient, and ensure that reads complete in two communication rounds if readers do not modify the storage, and in one communication round otherwise.

Our results demonstrate that the complexity gap between unauthenticated and authenticated storage can be effectively bridged without strongly limiting the adversary, providing answers to questions **Q4.1** and **Q4.2**.

6.1 Introduction

As already motivated in Chapter 1, robust storage implementations are attractive because they do not incur the overhead of cryptography and they are invulnerable to cryptographic attacks. However, existing robust algorithms with optimal resilience and optimal time complexity [GV06, GLV06, ACKM06, GV07] have a much higher read latency in the worst case, when compared to algorithms storing self-verifying data [MR98, CT06, LR06].

Read latency is critical because in many real world applications workloads are read-dominated. Therefore, it is natural to ask if it is possible to bridge the complexity gap between unauthenticated and authenticated distributed

storage without trading the valuable properties of robust storage.

In this chapter we show that this is indeed possible by describing robust storage implementations with optimal resilience and reduced optimal time complexity. Our algorithms circumvent the lower bounds established in [GV06, ACKM06] by using *secret tokens*. A secret token (briefly token) is a value randomly selected by the client and attached to the messages sent to the base objects.

The secrecy property of a token selected by a correct client is that the adversary can not generate its value before the client actually uses the token. Obviously, the assumption that tokens are secret can be violated with some probability. However, this probability can be arbitrarily reduced, for example, by uniformly and independently generating random tokens of k bits and by increasing the value of k. Note that in practice, assumptions in general hold only with a certain probability, e.g., the assumption that no more than t base objects fail.

Secret tokens are useful because they prevent faulty base objects from simulating client operations (read or write) that have not yet been invoked but will actually occur at some later point. However, tokens are weaker than signatures, because they cannot prevent a faulty base object from successfully forging a value that is never written. Roughly speaking, tokens help disambiguating (a) executions in which a read is contending with a write, and thus is allowed return an older value, from (b) executions in which the read is succeeding the write, and thus must not return an older value.

An alternative approach to the use of secret tokens to reduce the time complexity is the use of cryptography, namely digital signatures [MR98, CT06, LR06]. Digital signatures generally require the generation of a secret (e.g. private) key, which entails the generation of a random bit string.

Secret tokens have the following advantages over signatures: (1) no certification and key pre-distribution/sharing is needed, eliminating the need for a PKI and/or a trusted dealer; (2) no unproven assumptions such as the hardness of factorization or of discrete logarithm computation are needed; (3) the assumption of a computationally bounded adversary is not needed; (4) sampling of secret tokens can be done offline or asynchronously, without imposing an overhead in the critical execution path of the algorithm as done if signatures are used. Our algorithms are also designed to gracefully degrade their properties if the secrecy of the tokens is violated, whereas existing authenticated protocols do not discuss the system behaviors if signatures can be forged by the adversary.

6.1.1 Contributions

(1) We show that secret tokens can be used to reduce the read complexity of unauthenticated storage with optimal resilience from $t + 1$ communication rounds [ACKM06], to just two. The developed algorithm supports a possibly unbounded number of malicious readers. Moreover, our implementation is gracefully degrading. Even if the secrecy of tokens is violated, the algorithm preserves the safety properties of regular storage.

(2) We show that if readers do not write, then the cost of two communication rounds of the read operation is a lower bound for every unauthenticated storage algorithm with optimal resilience. Thus, the time complexity of our first algorithm is optimal. Notably, the lower bound of [ACKM06] does not hold in a model that allows the use of secret tokens.

(3) Under the assumption that readers can modify the base objects, we exhibit an implementation in which every read completes after one communication round. The read lower bound of two communication rounds [GV06] is circumvented by having readers store timestaped secret tokens in the base objects. This algorithm is also gracefully degrading. It preserves wait-freedom and never returns a forged value. It may however return an outdated value if the secrecy of tokens is violated.

6.2 Model

Following the definitions in Chapter 2, our model consists of a collection of clients, interacting with a finite collection of n base objects. Clients are divided into a singleton writer and a (possibly unbounded) set of reader processes. When needed, the number of readers is denoted by R. Up to $t \leqslant \lfloor \frac{n-1}{3} \rfloor$ base objects can fail by being nonresponsive-arbitrary (NR-Arbitrary) [JCT98]. Any number of reader processes can suffer Byzantine failures and the writer may fail by crashing. Clients interact with the base objects by message-passing using point-to-point reliable channels. Base objects do not communicate with each other and do not push messages to clients.

We assume the existence of a function GetToken used by clients that takes no arguments and outputs a value in $\{0, 1\}^*$ and has the following property:

Secrecy: The adversary cannot generate the i^{th} output of function GetToken before the i^{th} invocation of GetToken.

This assumption can be implemented by sampling a value (called token) randomly, uniformly and independently from $\{0,1\}^k$. With 2^k different tokens and large k (in practice a few bytes suffice), the probability of creating a token before learning it is negligibly small.

6.3 An Implementation Supporting Unbounded Readers

Our first algorithm uses $n \geqslant 3t+1$ base objects to implement a multi-reader single-writer (MRSW) regular storage and features optimal time complexity for both operations (see Section 6.3.4). In the following we give a detailed description of the algorithm.

6.3.1 Overview

Both READ and WRITE operations take at most two rounds. In each round, the client sends a message to all objects. Each round terminates at the latest after receiving matching replies from $n-t$ correct objects. A value is written in two consecutive phases, called *pre-write* and *write* phase. In the first READ round, the reader samples a set of candidates such that the value returned after the second round is among them. In the second round, the reader collects from the objects copies of the values in the candidate set, until it finds a value to return.

The base objects maintain the array $history[0\dots]$ used by the base objects to keep track of the values written. The entry $history[ts].pw$ stores a timestamp-value pair $tsval$ of the form $\langle ts, v\rangle$ and $history[ts].w$ the pair $\langle tsval, token\rangle$. The initial token value is the empty token denoted ϵ. Variable ts stores the timestamp of the last written value. The variables of an object are collectively called *fields*.

In the pre-write phase, of WRITE(v), the writer: (1) increases its timestamp ts, (2) assigns the timestamp-value pair $\langle ts, v\rangle$ to its variable pw and (3) writes pw to $n-t$ objects' $history[ts].pw$ field (short pw field). In the write phase, the writer (1) saves the previously written value w in the variable w_p, (2) invokes GetToken and assigns its output to variable $w.token$, (3) assigns pw to $w.tsval$ and (4) writes both w and w_p to $n-t$ objects' $history[ts].w$ and $history[ts-1].w$ fields respectively (short w fields). The algorithms of the writer and the base objects appear in Figures 6.1 and 6.2 respectively.

In the following we detail the READ implementation since it is more involved and constitutes the main focus of this chapter.

```
    Initialization:
1       ts ← 0; w ← ⟨⟨0, v_0⟩, ϵ⟩
    WRITE(v)
        /* Pre-write Phase */
2       inc(ts)
3       pw ← ⟨ts, v⟩
4       send pw⟨ts, pw⟩ to all objects
5       wait for reception of pw_ack⟨ts⟩ from n − t objects
        /* Write Phase */
6       w_p ← w
7       w.token ← GetToken()
8       w.tsval ← pw
9       send wr⟨ts, w, w_p⟩ to all objects
10      wait for reception of wr_ack⟨ts⟩ from n − t objects
11      return ack
```

Figure 6.1: Algorithm of the writer.

```
    Initialization:
1       ts ← 0; history[0].pw ← ⟨0, v_0⟩; history[0].w ← ⟨pw, ϵ⟩
2   upon reception of pw⟨ts′, pw⟩ from writer
3       history[ts′].pw ← pw
4       send pw_ack⟨ts′⟩ to writer
5   upon reception of wr⟨ts′, w, w_p⟩ from writer
6       if ts′ > ts then ts ← ts′
7       history[ts′].w ← w; history[ts′ − 1].w ← w_p
8       send wr_ack⟨ts′⟩ to writer
9   upon reception of rd1⟨tsr⟩ from reader j
10      send rd1_ack⟨tsr, history[ts].w, history[ts − 1].w⟩ to reader j
11  upon reception of rd2⟨tsr, TS⟩ from reader j
12      PW ← {history[ts′].pw : ts′ ∈ TS}
13      W ← {history[ts′].w : ts′ ∈ TS}
14      send rd2_ack⟨tsr, PW, W⟩ to reader j
```

Figure 6.2: Algorithm of the base objects.

6.3.2 READ Implementation

The full algorithm of the readers can be found in Figure 6.3. As mentioned earlier, READ performs in two rounds. In the first round, the reader collects

from $n-t$ base objects the latest and the second latest values written w and w_p and adds them to the set of return candidates C. For this purpose the reader sends a message *rd1* to all objects (line 4) and awaits $n-t$ matching responses of type *rd1_ack* (line 6).

```
    Predicates:
        safe(c) ≜ |{i ∈ Q : c.tsval ∈ PW[i] ∨ c ∈ W[i]}| ⩾ t + 1
        incomplete(c) ≜ |{i ∈ Q : c ∉ W[i]}| ⩾ n − t
        highCand(c) ≜ c ∈ C : (∀c′ ∈ C : c.tsval.ts ⩾ c′.tsval.ts)
    READ()
1       C ← TS ← Q ← ∅
2       PW[i] ← W[i] ← ∅, 1 ⩽ i ⩽ n
        /* Round 1 */
3       inc(tsr)
4        send rd1⟨tsr⟩ to all objects
        repeat
5           if received rd1_ack⟨tsr, w, w_p⟩ then C ← C ∪ {w, w_p}
6       until received rd1_ack⟨tsr, ∗⟩ from n − t objects
7        TS ← {c.tsval.ts : c ∈ C}
        /* Round 2 */
8        send rd2⟨tsr, TS⟩ to all objects
        repeat
9           if received rd2_ack⟨tsr, PW, W⟩ from object i then
10              Q ← Q ∪ {i}; PW[i] ← PW; W[i] ← W
11              C ← C\{c ∈ C : incomplete(c)}
12      until (received rd2_ack⟨tsr, ∗⟩ from n − t objects) ∧
              (∃c ∈ C : safe(c) ∧ highCand(c))
13      return c.tsval.val
```

Figure 6.3: Algorithm of the readers.

In the second round, the reader gathers copies of the candidate values in C from the history of pw and the w fields of the base objects until it finds a candidate it can safely return. For this purpose, in the second round (1) the reader adds the timestamps of the candidates in C to a set TS (line 7) and (2) sends a message *rd2* to all objects (line 8). Upon reception of a *rd2* message, each correct object constructs two sets PW and W, and for each timestamp $ts \in TS$ it adds to PW and W the corresponding value from the $history[ts].pw$ and $history[ts].w$ fields, if present. Finally, it sends a *rd2_ack* message containing PW and W back to the reader. When the reader receives a matching *rd2_ack* message from base object i for the first time, it records

PW and W in its variables $PW[i]$ and $W[i]$, and removes all candidates from C which are incomplete (lines 9–11). If a value c is incomplete then it is missing from $n - t$ objects' history of w fields. In this case, the WRITE of c does not precede READ and thus c can be disregarded without violating regularity. The reader keeps waiting for additional *rd2_ack* messages until there is a candidate $c \in C$ such that no candidate in C has a higher timestamp (i.e., predicate highCand(c) holds) and c is stored at $t + 1$ base objects in the pw or w field (i.e., predicate safe(c) holds).

Our implementation guarantees that the condition in line 12 is eventually satisfied in every READ. In the following we give a rough intuition of why this is true (the detailed proof can be found in Section 6.3.3).

Observe that $C \neq \varnothing$ because the second-last written value reported by a correct object is never incomplete. Assume by contradiction that READ never completes, i.e. there is a candidate $c \in C$ such that c is never eliminated from C and c is never safe. Consider the following two cases. Case (1): c is reported in the first READ round *after* the pre-write phase of $c.tsval$ has completed. In this case, $c.tsval$ is pre-written to $t + 1$ correct objects before any of them is accessed by the second READ round. Hence $t + 1$ correct objects eventually report $c.tsval$ from their pw history and c becomes safe. Case (2): c is reported during the first READ round *before* the pre-write phase of $c.tsval$ has completed. Clearly, c is reported by a malicious object. By the Secrecy assumption, the token used by the adversary is different from the token which is indeed written together with $c.tsval$. Hence, no correct object reports c and c is eliminated from C. Therefore, each value either becomes safe or is removed from the set of candidates.

It is important to note that the algorithm implements a regular storage even if the Secrecy assumption does not hold. Specifically, the proof of regularity below does not rely on the inability of the adversary to guess the token.

6.3.3 Correctness

Lemma 31 (Regularity)**.** *The* READ *operation either returns the latest value written before* READ *is invoked or one that is written concurrently with* READ*.*

Proof. Note that if READ returns a value $c.tsval.val$, then safe(c) holds. This implies that $t + 1$ objects respond with $c.tsval$ and some of these is correct. Hence, either $c.tsval$ has been written or is $\langle 0, v_0 \rangle$. We now show that READ does not return values older that the latest WRITE preceding READ.

If no WRITE completes before READ then we are done. Else, let R be a READ invocation and W = WRITE(v) be the last WRITE that completes

before R is invoked. Let ts be the timestamp associated with v. We need to show that if $c.tsval.val$ is returned, then $c.tsval.ts \geqslant ts$.

We assume by contradiction that $c.tsval.ts < ts$. Since W precedes R, the write phase of $\langle ts, v \rangle$ completes at $t+1$ correct objects before any of them is accessed by R. Therefore, these $t+1$ objects report to the first round of R values with timestamp ts or higher. Since READ waits for $n-t$ responses, it receives a response from one of these $t+1$ correct objects. Let i denote this object and let c' be the value with the lowest timestamp of the two values reported by i such that $c'.tsval.ts \geqslant ts$. We show that c' is not incomplete. Assume the contrary.

By definition of incomplete, c' is missing from the history of $n-t$ objects. There are two cases to consider. If c' is reported in w, then by the choice of c', it holds that $c'.tsval = \langle ts, v \rangle$. Otherwise, c' is reported in w_p, which implies that WRITE($c'.tsval.val$) precedes the second round of R. In both cases c' has been stored in the history of w fields of $t+1$ correct objects before the second read round starts. Hence, c' is missing from the history of w fields of at most $n-t-1$ objects, a contradiction. Consequently, c' is not incomplete and is never removed from the set C of candidates. As $c'.tsval.ts \geqslant ts > c.tsval.ts$, c is not highCand, contradicting the assumption that R returns $c.tsval.val$. □

Lemma 32 (Wait-freedom)**.** READ *and* WRITE *operations are wait-free.*

Proof. As the WRITE operation waits for at most $n-t$ objects to respond and by assumption there are $n-t$ correct objects, it never blocks. We now show that the READ operation does not block.

We assume by contradiction that READ blocks in line 23. We consider the time after which all correct objects (at least $n-t$) have responded. We first show that $C \neq \varnothing$. Let c be the second-last value written to a correct object and reported in w_p (line 5). Observe that WRITE($c.tsval.val$) is complete before the second round of R starts. Therefore c is missing from the history of at most $n-t-1$ objects and thus, c is never eliminated from C.

We now show that for all $c \in C$, safe(c) holds. Assume by contradiction that there exists $c \in C$ and c is not safe. We distinguish the following two cases:
Case (1): c is reported in the first round by some correct object. This implies that $c.tsval$ is pre-written to $t+1$ correct objects before any of them is read in the second round. Therefore, these $t+1$ correct objects respond with $c.tsval$ in PW and c is safe. Case (2): only malicious objects respond with c in the first read round. If no correct object reports c in the second read round, then c is incomplete and hence $c \notin C$. Else, if some correct object reports $c' = c$, then $c'.token = c.token$. By the Secrecy property, the malicious base

objects report c only after the WRITE of c' has invoked GetToken. As the pre-write phase precedes the invocation of GetToken, $c.tsval$ is pre-written to $t+1$ correct objects before the second READ round starts and therefore c is safe. □

Theorem 8. *The Algorithm appearing in figures 6.1, 6.2 and 6.3 wait-free implements a MRSW regular storage.*

Proof. Follows directly from Lemma 31 and Lemma 32. □

Efficiency After having proved the correctness, we now discuss the efficiency of the algorithm. As the algorithm stores the history of written values in the base objects, the storage requirements depend on the number of write operations. Note that, if readers do not write, storing less values is an open problem [CGK07]. The messages used are of constant size except the second read round messages which are $O(n)$. Observe that neither the storage requirements of the base objects nor the communication complexity (i.e. message size) depends on the number of readers in the system. Thus, the algorithm is scalable, supporting a possibly unbounded number of malicious clients. As announced, the time complexity of both READs and WRITEs is of two rounds in the worst case.

In the following we show that the round-complexity of the algorithm is tight.

6.3.4 Optimality: Fast Reads Must Write

In this section we show that the presented algorithm has optimal time complexity. This result, together with the lower bound of two rounds for the WRITE [ACKM06], imply that our algorithm exhibits optimal time complexity.

Intuitively, our proof derives from three indistinguishable runs. In the first run, READ is concurrent with WRITE, all correct base objects have responded and the faulty objects have crashed. In the second run, WRITE precedes READ but the faulty objects are malicious and hide the written value from the reader, simulating the concurrency of the first run. In the third run, no value is written and the malicious base objects forge the value of the writer. The reader finds itself in a situation in which it cannot distinguish between the second and the third run. If the reader returns a value, then it returns the same value in both runs, which violates safety either in the second or the third run. Else if the reader waits for more base objects, then it would block in the first run, which violates liveness.

The proof uses similar arguments as the lower bound proof in [GV06] and is illustrated in Figure 6.4. We partition the set of base objects into four distinct subsets T_1, T_2, T_3, T_4 each of size t. The initial state of every correct base object is denoted as σ_0. A round *rnd* of an operation is depicted by a line of rectangles. A rectangle in a line corresponding to some round *rnd* of operation *op* means that all base objects in the corresponding block have received the message from the client in round *rnd* of operation *op* and have sent a reply message.

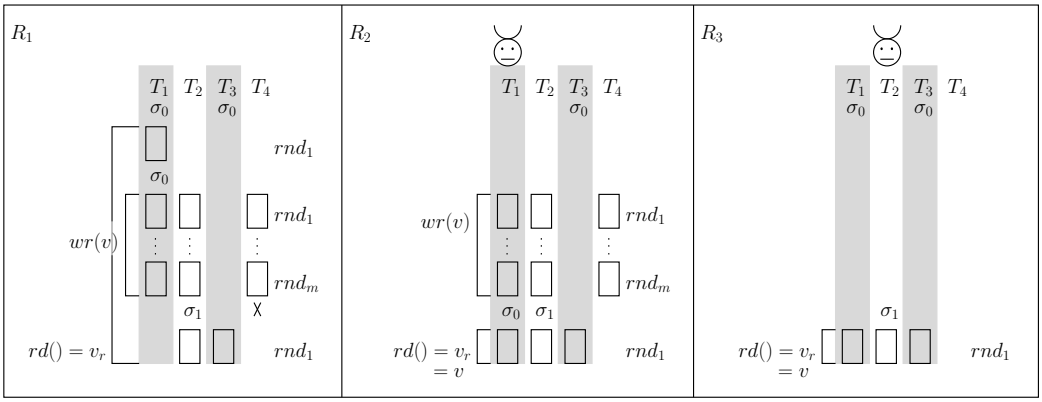


Figure 6.4: Illustration of the proof of Theorem 9

Theorem 9. *There is no fast* READ *implementation I of a SRSW safe storage from $4t$ base objects if the reader does not modify the base objects' state.*

Proof. To exhibit a contradiction, we construct a run of the safe implementation I that violates safety. We exhibit a run in which some READ returns a value that was never written.

- Let R_1 be the run in which all objects are correct except T_4 that crashes at a later point. Furthermore, let rd be the READ operation by reader r. After T_1 sends rd_ack to r, object T_1 is still in the initial state σ_0. Before rd reads from another object, a WRITE operation wr is invoked by the correct writer to write a value $v \neq v_0$ to the storage. By the assumption that I is wait-free, wr completes in R_1, say at time t_1 after invoking a finite number m of rounds. Due to asynchrony all messages sent by the writer to T_3 during w remain in transit. We refer to the state of base object T_2 at time t_1 as σ_1. At some time after t_1, object T_4 crashes. Due to asynchrony, all messages exchanged between r and T_2 and T_3 are delayed until after t_1. By our assumption on wait-freedom of

I, r completes after receiving *read_ack* messages from correct objects T_1, T_2, and T_3 and returns some value v_r skipping T_4.

- Let R_2 be the run similar to run R_1, except that in R_2: (1) READ rd is invoked only after wr completes (after t_1) and (2) object T_1 is malicious and at t_1 before replying to r, forges its state to σ_0, the initial state of correct objects. Other messages are delivered as in R_1. Note that wr cannot distinguish run R_2 from R_1 and therefore wr completes in R_2 at t_1. Note also that, rd is invoked after wr completes, so safety implies that rd must return v. However, note that in R_1 and R_2 the reader receives in rd the identical messages and, since the processes do not have access to a global clock, r cannot distinguish R_2 from R_1. Therefore, in R_1 and R_2, rd returns the same value, i.e. by safety v_r must equal v.

- Finally, we consider the run R_3 in which wr is never invoked, but T_2 is malicious and forges its state to σ_1 at the beginning of the run. READ rd is invoked in R_3 as in R_2. Since, upon receiving *read_ack* messages from T_1, T_2, and T_3, the reader receives identical information as in run R_2, the reader cannot distinguish R_2 from R_3, and rd completes in R_3 and returns $v_r = v$. However, by safety, in R_3, rd must return v_0. Since $v \neq v_0$, safety is violated in R_3.

□

6.4 An Implementation of Fast READs

The second presented algorithm also uses $n \geqslant 3t + 1$ base objects and implements a MRSW regular storage. The main difference to the previous algorithm is that every READ operation completes after one communication round.

6.4.1 Overview

In each round the client (reader or writer) sends a message to all objects and waits until it has received matching replies from at most $n - t$ correct objects. Like in the previous algorithm, a value is written in two phases, a pre-write and a subsequent write phase. Unlike in the previous algorithm, in the pre-write phase, in addition to writing data, the writer also reads control data from the base objects. Readers write control data and read data written by the writer.

The base objects maintain in addition to the history of written values an array $tsrtoken[1...r]$ which is updated by the readers. The entry $tsrtoken[j]$ stores a timestamp-token pair of the form $\langle tsr, token \rangle$, where tsr is the most recent timestamp of reader j and $token$ the corresponding token value.

In the pre-write phase, of WRITE(v), the writer: (1) increases its timestamp ts, (2) stores the last pre-written value in pw_p (3) assigns the timestamp-value pair $\langle ts, v \rangle$ to its variable pw, (4) writes pw and w to $n-t$ objects' $history[ts].pw$ and $history[ts-1].w$ fields respectively, (5) reads the objects' $tsrtoken[*]$ fields written by the readers and (6) for each reader j adds $tsrtoken[j]$ to the set $Tsrtokens[j]$. In the write phase, the writer (1) assigns $\langle pw, Tsrtokens \rangle$ to variable w and (2) writes both w and pw_p to $n-t$ objects' $history[ts].w$ and $history[ts-1].pw$ fields respectively. The algorithm of the writer appears in Figure 6.5.

In the following we detail the READ implementation and the interaction with the base objects, which is slightly more involved.

```
   Initialization:
1      Inittsrtokens[j] ← ∅, 1 ⩽ j ⩽ r
2      ts ← 0; pw ← ⟨0, v_0⟩; w ← ⟨pw, Inittsrtokens⟩

   WRITE(v)
       /* Pre-Write Phase */
3      Tsrtokens ← Inittsrtokens
4      inc(ts)
5      pw_p ← pw
6      pw ← ⟨ts, v⟩
7      send pw⟨ts, pw, w⟩ to all objects
       repeat
8          if received pw_ack⟨ts, tsrtoken⟩ from object i then
9              Tsrtokens[j] ← Tsrtokens[j] ∪ {tsrtoken[j]}, 1 ⩽
               j ⩽ r
10     until received pw_ack⟨ts, *⟩ from n − t objects
       /* Write Phase */
11     w ← ⟨pw, Tsrtokens⟩
12     send wr⟨ts, pw_p, w⟩ to all objects
13     wait for reception of wr_ack⟨ts⟩ from n − t objects
14     return ack
```

Figure 6.5: Algorithm of the writer.

6.4.2 READ Implementation

The full algorithm of the base objects is given in Figure 6.6 and that of the readers in Figure 6.7. As mentioned earlier, READ completes in one communication round. The reader (1) increments its timestamp tsr, (2) selects a secret token $token$ and (3) sends a message rd containing tsr and $token$ to all objects. Upon reception of rd from reader j, each correct object (1) stores $\langle tsr, token \rangle$ in $tsrtoken[j]$, (2) computes a timestamp ts_{max} such that any higher timestamped value stored has been written concurrently with READ and (3) sends a message rd_ack containing three values with timestamps $ts_{max} - 1$, ts_{max} and $ts_{max} + 1$ (if available) back to the reader. When the reader receives a rd_ack message from object i for the first time, it stores the value with timestamp ts_{max} in $w[i]$ and adds $w[i]$ to the set of candidates C. The other two values are added to $PW[i]$. In addition it removes all incomplete candidates from C. A candidate is incomplete when $n - t$ objects have reported candidates with lower timestamps. Observe that the choice of ts_{max} as candidate is crucial: (a) values with higher timestamps can be safely disregarded without violating regularity and (b) the value corresponding to ts_{max} is stored in $t + 1$ correct objects' pw field before any of them is read. The latter property is critical because otherwise, a candidate might never become safe. The termination condition is the existence of a candidate which is both highCand and safe. Our implementation guarantees that this condition is eventually satisfied in every READ. We now give an intuition of why this is true.

Recall that, for every candidate c it holds that c is pre-written to $t + 1$ correct objects before any of them is read. We now explain why. The negation thereof implies that at least $t + 1$ correct objects store the timestamp-token pair of READ *before* c is pre-written to them. At least one of them reports the token in the pre-write phase, such that c and all higher timestamped values are stored together with the token in the write phase. Consequently, all correct objects (at least $n - t$) report to READ only values with lower timestamps and c is eliminated from C. It is not difficult to see that if the correct base objects report the entire pw history, then every candidate would eventually become safe. Our approach simulates this behavior, but the correct objects send at most three values, with consecutive timestamps centered around ts_{max}. The reasoning behind it is the following: if some candidate is lower than the first, then it is not highCand. Else, if it is higher than the third, then it is removed from C.

Initialization:

1 $Inittsrtokens[j] \leftarrow \varnothing$; $tsrtoken[j] \leftarrow \langle 0, \epsilon \rangle$, $1 \leqslant j \leqslant r$
2 $history[0].pw \leftarrow \langle 0, v_0 \rangle$;
$history[0].w \leftarrow \langle \langle 0, v_0 \rangle, Inittsrtokens \rangle$

3 **upon** reception of $pw\langle ts, pw, w \rangle$ from writer
4 $history[ts].pw \leftarrow pw$; $history[ts-1].w \leftarrow w$
5 send $pw\_ack\langle ts, tsrtoken \rangle$ to writer

6 **upon** reception of $wr\langle ts, pw_p, w \rangle$ from writer
7 $history[ts-1].pw \leftarrow pw_p$; $history[ts].w \leftarrow w$
8 send $wr\_ack\langle ts' \rangle$ to writer

9 **upon** reception of $rd\langle tsr, token \rangle$ from reader j
10 **if** $tsr > tsrtoken[j].tsr$ **then** $tsrtoken[j] \leftarrow \langle tsr, token \rangle$
11 $ts_{max} \leftarrow \max\{ts : tsrtoken[j] \notin history[ts].w.Tsrtokens[j]\}$
12 $w \leftarrow history[ts_{max}].w.tsval$
13 $PW \leftarrow \{history[ts_{max}-1].pw, history[ts_{max}+1].pw\}$
14 send $rd\_ack\langle tsr, PW, w \rangle$ to reader j

Figure 6.6: Algorithm of the base objects.

Predicates:

$\mathsf{safe}(c) \triangleq |\{i \in Q : c \in PW[i] \cup \{w[i]\}\}| \geqslant t+1$
$\mathsf{incomplete}(c) \triangleq |\{i \in Q : w[i].ts < c.ts\}| \geqslant n-t$
$\mathsf{highCand}(c) \triangleq \forall c' \in C : c.ts \geqslant c'.ts$

READ()

15 $C \leftarrow Q \leftarrow \varnothing$
16 $PW[i] \leftarrow \varnothing$; $w[i] \leftarrow \perp$, $1 \leqslant i \leqslant n$
17 $inc(tsr)$
18 $token \leftarrow \mathsf{GetToken}()$
19 send $rd\langle tsr, token \rangle$ to all objects
repeat
20 **if** received $rd\_ack\langle tsr, PW, w \rangle$ from object i **then**
21 $Q \leftarrow Q \cup \{i\}$; $PW[i] \leftarrow PW$; $w[i] \leftarrow w$; $C \leftarrow C \cup \{w\}$
22 $C \leftarrow C \backslash \{c \in C : \mathsf{incomplete}(c)\}$
23 **until** (received $rd\_ack\langle tsr, * \rangle$ from $n-t$ objects) $\wedge$
$(\exists c \in C : \mathsf{safe}(c) \wedge \mathsf{highCand}(c))$
24 **return** $c.val$

Figure 6.7: Algorithm of the readers.

6.4.3 Correctness

Lemma 33 (Regularity)**.** *The* READ *operation either returns the latest value written before* READ *is invoked or one that is written concurrently with* READ*.*

Proof. Observe that if READ returns a value $c.val$, then $\mathsf{safe}(c)$ holds. This implies that $t+1$ objects respond with c and some of these is correct. Hence, either c has been written or is $\langle 0, v_0 \rangle$. We now show that READ does not return values older than the latest WRITE preceding READ.

If no WRITE completes before READ then we are done. Else, let R be a READ invocation of reader j and W = WRITE(v) be the last WRITE that completes before R is invoked. Let ts be the timestamp associated with v. We need to show that if $c.val$ is returned, then $c.ts \geqslant ts$.

We assume by contradiction that $c.ts < ts$. Let $\langle tsr, token \rangle$ be the timestamp-token pair of R. Since W precedes R, GetToken is invoked by R after W is complete. If some malicious object reports $\langle tsr, token' \rangle$ to W, then the Secrecy assumption implies that $token \neq token'$. Therefore, W does not include $\langle tsr, token \rangle$ in the set $Tsrtokens[j]$ corresponding to $\langle ts, v \rangle$. Furthermore, the write phase of $\langle ts, v \rangle$ completes at $t+1$ correct base objects before any of them is accessed by R. As $\langle tsr, token \rangle \notin Tsrtokens[j]$, these $t+1$ correct objects report values with timestamp ts or higher from their w field.

Let c' be the value with the lowest timestamp received from the w field of any of the $t+1$ objects. As R waits for at least $n-t$ objects to respond, such a value exists. We show that c' is not incomplete. Assume the contrary. By definition of incomplete, $n-t$ objects must report values with timestamps lower than $c'.ts$ from their w field. At least one of these is a correct object i among the $t+1$ updated by W. By the choice of c', $w[i].ts \geqslant c'.ts$. Therefore, c' is not incomplete and is never removed from the set C of candidates. As $c'.ts \geqslant ts > c.ts$, c is not highCand, contradicting the assumption that c is returned by R. □

Lemma 34 (Wait-freedom)**.** READ *and* WRITE *operations are wait-free.*

Proof. As the WRITE operation waits for at most $n-t$ objects to respond and by assumption there are $n-t$ correct objects, it never blocks. We now show that the READ operation does not block.

We assume by contradiction that READ blocks in line 23. We consider the time after which all correct objects (at least $n-t$) have responded. We first show that $C \neq \varnothing$. Let c be the $(t+1)$th highest value reported in the w field of a correct object. Clearly, c is not incomplete and thus it is not removed from C.

Let $c \in C$ be the highest value reported in the w field of a correct object. We show that (1) $\mathsf{highCand}(c)$ holds and (2) $\mathsf{safe}(c)$ holds.

Step (1): If c is not highCand, then there exists $c' \in C$ and $c'.ts > c.ts$. By the choice of c, there are $n - t$ correct objects i that report values $w[i]$ such that $w[i].ts < c'.ts$. This implies that $c' \notin C$, a contradiction.

Step (2): Observe that $t+1$ correct objects have stored c in their pw field before any of them replies to READ. Else, no correct object would reply with c in the w field (line 11). Let i be any of these correct objects. We assume by contradiction that $c \notin PW[i] \cup \{w[i]\}$. Let ts_{max} be the timestamp computed by object i in line 11. If $c.ts - 1 \leqslant ts_{max} \leqslant c.ts + 1$, then c is reported either from $PW[i]$ or $w[i]$ and we are done. Observe that, since c is pre-written to i together with the last written value (with timestamp $c.ts - 1$), it holds that $ts_{max} \geqslant c.ts - 1$. Therefore, the only remaining case is $ts_{max} > c.ts + 1$. This implies that c' exists such that $ts_{max} > c'.ts > c.ts$. Since the value with timestamp ts_{max} is pre-written to $t + 1$ correct objects before they are read, c' is written to $t + 1$ correct objects before they are read. Hence, c' or a higher timestamped value is not incomplete, contradicting the assumption that c is highCand. □

Theorem 10. *The Algorithm appearing in figures 6.5, 6.6 and 6.7 wait-free implements a MRSW regular storage.*

Proof. Follows directly from Lemma 33 and Lemma 34. □

Efficiency We now discuss the efficiency of the algorithm. Like in the previous algorithm, the storage requirements depend on the number of write operations. In addition, the base objects store up to $n \cdot r$ timestamp-token pairs together with each value written to them. Messages exchanged between the reader and the base objects are of constant size. The WRITE messages *pw_ack* (respectively *wr*) contain r (respectively $n \cdot r$) timestamp-token pairs. The time complexity of the READ is one communication round in the worst case, which is clearly optimal. Every WRITE completes after two rounds which is also optimal [ACKM06].

6.5 Summary of the Results

The presented robust algorithms effectively circumvent lower bounds established for unauthenticated storage by using secret tokens. The first algorithm

supports unbounded readers and features constant read complexity. The second algorithm features fast reads, i.e., every read terminates after one round of communication with the base objects. Even if the secrecy assumption of the token is violated both algorithms are gracefully degrading. The first algorithm fully preserves regularity and the second algorithm never blocks and never returns a forged value. However, the probability of property violation is negligibly small if the token space is large enough. The algorithms are secure against a computationally unbounded adversary because tokens are purely random and therefore they cannot be computed.

Chapter 7

Complexity of Robust Atomic Storage

In this chapter, we determine the time complexity of robust *atomic* storage from passive storage components prone to Byzantine faults. Robustness here means wait-free tolerating the largest possible number t of Byzantine base object failures (optimal resilience) without relying on data authentication. We show that no multi-reader robust atomic storage implementation exists if (a) read operations complete in less than *four* communication rounds, and (b) the time complexity of write operations is constant.

More precisely, we present two lower bounds. The first is a read lower bound stating that three rounds of communication are necessary to read from a multi-reader robust atomic storage. The second is a write lower bound, showing that $\Omega(\log(t))$ write rounds are necessary to read in three rounds from such a storage, answering question **Q5**.

Applied to known results, our lower bounds close a fundamental gap: time-optimal robust atomic storage can be obtained using well-known transformations from regular to atomic storage and existing time-optimal regular storage implementations.

7.1 Introduction

Variable sharing is critical to modern distributed and concurrent computing. The *atomic* read/write register abstraction [Lam86] is essential to sharing information in distributed systems. It abstracts away the complexity incurred by concurrent access to shared data by providing processes an illusion of sequential access to data. This abstraction is also referred to as atomic *storage*, for its importance as a building-block in practical distributed storage

and file systems (see e.g., [SH02, SFV+04]). Besides, its read/write API, despite being very simple, is today the heart of modern "cloud" key-value storage APIs (e.g., [AWS]).

In this chapter, we study atomic storage implementations in asynchronous message-passing systems in which a set of *clients* share data leveraging a set of storage *objects* processes subject to Byzantine faults. We consider fault-tolerant, *robust* storage implementations characterized by: a) wait-freedom [Her91] (i.e., the fact that read/write operations invoked by correct clients always eventually return) and b) optimal resilience, without assuming self-verifying data [MR98] to limit the adversary (by relying on e.g., digital signatures). Recall that, in the Byzantine failure model, optimal resilience corresponds to using $3t + 1$ base objects to tolerate t failures [MAD02].

In this model, we ask a fundamental question: what is the optimal worst-case time-complexity of robust atomic storage implementations?

Perhaps surprisingly and despite the wealth of literature exploring latency-optimal storage implementations, this question has not been answered. It is known that the worst-case latency of *writing* into robust storage is at least 2 rounds [ACKM06]. In this chapter, we show that the optimal worst-case latency of *reading* from *scalable* robust atomic storage is 4 (four) rounds. Here, the notion of scalability captures two basic criteria: a) support for any number of readers, and b) constant write-latency. Our results close a fundamental gap, showing that latency-optimal scalable and robust atomic storage, combining 2-round writes and 4-round reads, can be achieved (in the case of multi-reader single-writer (MRSW) storage) using standard transformations from weaker, regular [Lam86] registers to the atomic ones [Lyn96, AW98].

Our contribution goes through proving two lower bounds. To help fully appreciate our contributions, we first discuss how the scope of this thesis chapter fits into related work.

7.1.1 Previous and Related work

Several papers have explored the time-complexity metric in the context of a read/write register abstraction. A seminal crash-tolerant robust atomic MRSW register implementation assuming a majority of correct processes was presented in [ABD95]. In [ABD95], all write operations complete in a single round; on the other hand, read operations always take two rounds between a client and objects.

The problem of modifying [ABD95] to enable single round reads was explored in [DGLC04], which showed that such *fast* atomic implementations

are possible albeit they come with the price of limited number of readers and suboptimal resilience. Moreover, the reader in [DGLC04] needs to write (i.e., modify the objects' state) as dictated by the lower bound of [FL03] which showed that every atomic read must write into at least t objects. [DGLV05] extends the result of [DGLC04] to the Byzantine failure model assuming authenticated (i.e., digitally signed) data and established the impossibility of fast crash-tolerant multi-reader multi-writer (MRMW) atomic register implementations. This result is in line with classical MRMW implementations such as [LS02] that have read/write latency of at least 2 rounds. The limitation on the number of readers of [DGLC04], was relaxed in [GNS09], where a crash-tolerant robust MRSW atomic register implementation was presented, in which most of the reads complete in a single round, yet a fraction of reads is permitted to be slow and complete in 2 rounds.

In the Byzantine context, optimizing latency is particularly interesting when data is assumed to be unauthenticated, which we also assume here. [ACKM06] showed that any Byzantine-tolerant storage employing at most $4t$ storage objects has at least some write operation complete in 2 rounds. Moreover, [ACKM06] showed a tight lower bound of $t+1$ rounds from reading from robust MRSW safe [Lam86] storage, with the constraint that readers are precluded from writing. However, allowing readers to write helps improve latency as shown in [GV06], through a 2-round tight lower bound on reading from robust MRSW *regular* [Lam86] storage. This bound was circumvented in Chapter 6, assuming secret values used to detect concurrent operations, where reads are expedited to complete in a single round. However, none of these papers dealt with optimal worst-case latency of reading from robust *atomic* storage, which is precisely the scope of this chapter.

On the other hand, few papers have explored the *best-case* complexity of Byzantine-tolerant optimally resilient atomic storage. Here, "best-case" encompasses synchrony, no or few object failures and the absence of read/write concurrency. In this context, [GLV06] presented the first robust atomic storage implementation in which both reads and writes are fast in the best-case (i.e., complete in a single round-trip). Furthermore, [GV07] considered robust atomic storage implementations with the possibility of having fast reads and writes gracefully degrade to 2 or 3 rounds, depending on the size of the available quorum of correct objects. Unlike these papers, we are interested here with the unconditional, *worst-case* latency of atomic storage.

Finally, the worst-case read latency in existing Byzantine-tolerant robust atomic storage implementations for unauthenticated data (e.g., [MAD02, GLV06, GV07, AAB07]) is either unbounded or $\Omega(t)$ rounds at best [AAB07].

7.1.2 Contributions

We present two lower bounds (impossibility results) on time-complexity of reading from robust atomic storage for unauthenticated data, implemented from storage objects prone to Byzantine faults. Together, our lower bounds imply that *there is no* scalable robust atomic storage implementation in the Byzantine unauthenticated model in which all reads complete in less than 4 rounds.

- The first lower bound, referred to as the *read lower bound*, demonstrates the impossibility of reading from robust MRSW atomic storage in two rounds. More precisely, we show that if the number of storage objects S is at most $4t$ and if the number of readers R is greater than 3, then no MRSW atomic implementation may have all reads complete in two rounds.

  Our proof scheme resembles that of [DGLC04] and relies on sequentially appending reads on a write operation, while progressively deleting the steps of a write and preceding read operations, exploiting asynchrony and possible failures. This deletion ultimately allows reusing readers and reaching an impossibility with as few as $R = 4$ readers. As none of these appended operations are concurrent under step contention, the impossibility also holds in the stronger unauthenticated data model augmented with secrets (Chapter 6), in which the adversary is unable to simulate step contention among operations, making use of secret values.

- Our second lower bound, referred to as the *write lower bound*, shows that if read operations are required to complete in three communication rounds, then the number of write rounds k is $\Omega(log(t))$. More precisely, we show that if the number of storage objects is at most $3t + \lfloor t/t_k \rfloor$ and $R \geqslant k$, then no implementation of a MRSW atomic storage may have all reads complete in three rounds and all writes in $k \leqslant \lfloor \log(\lceil \frac{3t_k+1}{2} \rceil) \rfloor$ rounds. In a sense, our lower bound generalizes the write lower bound of [ACKM06], which proves our result for the special case of $k = 1$.

  While using a similar approach, the write lower bound proof is much more involved and differs from our read lower bound proof in several key aspects. Due to the additional third read round, read steps cannot be entirely deleted, which prohibits the reuse of readers. Consequently, the number of supported readers R and the number of write rounds k are related ($R \geqslant k$). Furthermore, the proof relies on a set of malicious objects that forges critical steps of the write and of prior reads with

respect to subsequent reads. This set grows with the number of appended reads, relating the number of faulty objects t and the number of readers (which is at least k). At the heart of the proof we use a recurrent formula that relates t and k, similar to a Fibonacci sequence, which describes the exact relation between the two parameters. In its closed form, the formula transforms to the log function ($k = \Omega(log(t))$).

The rest of the chapter is organized as follows. In Section 7.2 we give our model, and Section 7.3 gives the proof of our read lower bound. An extension to the model of [TP88] using distinct thresholds for malicious and crash objects' failures is included in Appendix B. Section 7.4 gives the proof of our write lower bound. Section 7.5 concludes the chapter by discussing modular implementations that match our lower bounds.

7.2 Model

The model considered herein is in line with the message-passing model formally defined in Chapter 2, which we now briefly reiterate. There are three *disjoint* sets of processes: a set *objects* of size n containing processes $\{s_1, ..., s_n\}$ and representing the base register elements; a singleton writer containing a single process $\{w\}$; and a set *readers* of size R containing processes $r_1, ..., r_R$. The set clients is the union of the sets writer and readers. We assume that every client may communicate with any process by message passing using point-to-point reliable communication channels. However, objects cannot communicate among each other, nor send messages to clients other than in reply to clients' messages.

7.3 The Read Lower Bound

In this section we prove the following proposition.

Proposition 1. *: If $n \leqslant 4t$ and $R > 3$, then no* read *implementation I of a multi-reader (MRSW) atomic register exists that completes in two rounds.*

Overview

The idea behind the proof is to start with a complete write that writes 1 into the storage, after which a complete read is appended. By atomicity, the read returns 1. Then, further reads by distinct readers are appended one after the other such that the last appended read returns 1. At the same time,

steps of the write and the previous reads are progressively deleted. After appending the fourth read, the final round of the write is deleted from the storage. Moreover, similar to a circular buffer, all steps of the first read are erased, and the read ca be "recycled". By atomicity, the last appended read returns 1. The next iteration starts by reusing the first read, which in turn frees the second read. The proof proceeds through a sequence of such iterations. In each iteration, the last appended read frees the first appended read, and deletes another round of the write. After the last iteration, all steps of the write are deleted, meaning that no write is invoked. However, the last appended read returns 1, violating atomicity.

Preliminaries

In the proof w denotes the writer, r_i for $1 \leqslant i \leqslant 4$ denote the readers, and s_i for $1 \leqslant i \leqslant n$ denote objects. Suppose by contradiction that $R = 4$ and there is an atomic register implementation I that uses at most $4t$ objects, such that in every partial run of I every *read* operation completes in two rounds.

We partition the set *objects* into four disjoint subsets (which we call *blocks*), denoted B_i for $1 \leqslant i \leqslant 4$ each of size exactly $t \geqslant 1$. We refer to the initial state of every correct block B_j as σ_0^j. For simplicity we simply write σ_0, where the block name is implicit.

We say that a round *rnd* of an operation *op skips* a set of blocks BS in a partial run, (where $BS \subseteq \{B_1, \ldots, B_4\}$), if (1) no object in any block $BL \in BS$ receives any message in round *rnd* from *op* in that partial run; (2) all other objects receive all messages in round *rnd* from *op* and reply to the messages, and (3) in case round *rnd* is terminated, the invoking client has received all these reply messages or, in case *rnd* is not terminated, all these reply messages are in transit. We say that an operation *op* skips a set of blocks BS in a partial run if every round of *op* skips BS.

To show a contradiction, we construct a partial run of the implementation I that violates atomicity: a partial run of I in which no value is ever written and some read returns 1.

Partial writes

Throughout the proof there is only one write operation write(1) by w that writes value 1. Consider a partial run wr in which w completes write(1) on the register and let k be the number of rounds invoked by w in wr. We denote the state of every correct block B_j after it has replied to the messages of the write during round 1 to i where $1 \leqslant i \leqslant k$ as σ_i, where j is again implicit. The write operation skips blocks B_4. We define a series of partial

runs containing an incomplete write(1) invocation, each being a prefix of wr. For $1 \leqslant i \leqslant k$ and $1 \leqslant j \leqslant 4$, we define wr_j^i as the partial run in which (1) rounds 1 to $i-1$ are terminated and skip B_4; (2) round i is not terminated and skips all blocks $\{B_l \mid 1 \leqslant l \leqslant j-1\} \cup \{B_4\}$, and (3) all objects are correct. We make two observations: (1) partial run wr_1^k differs from wr only at w and (2) partial run wr_4^1 differs from a run in which write(1) is never invoked only at w.

Block diagrams

We illustrate the proof in Figure 7.1 (a)-(g). We depict a round rnd of an operation op through a set of rectangles arranged in a single column. In the column corresponding to some round rnd of op we draw a rectangle in a given row, if all objects in the corresponding block BL have received the message from the client in round rnd of op and have sent reply messages, i.e., if round rnd of op does not skip BL. We write "@" in the row corresponding to BL iff BL is malicious.

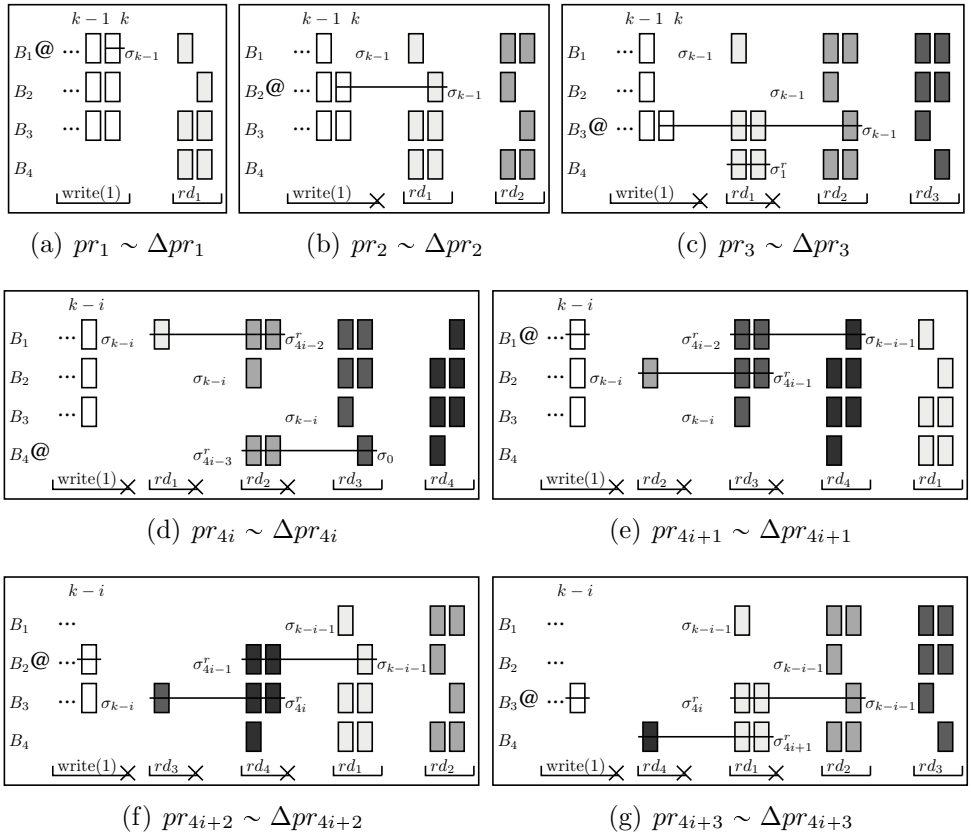


Figure 7.1: Illustration of the runs used in the proof of Proposition 1

Appending reads

Partial run pr_1 extends wr by appending a complete read rd_1 by r_1 that skips B_2 in round one and B_1 in round two (see Figure 7.1 (a)). Note that when the second round is started, there is a pending first round invocation on B_2. Therefore in the second round, rd_1 waits for both first *and* second round replies from B_2. For ease of presentation, the late first round replies are not illustrated.

In pr_1, all objects in block B_1 are malicious, and forge their state to σ_{k-1} before replying to rd_1. By atomicity rd_1 returns 1. Observe that r_1 cannot distinguish pr_1 from some partial run Δpr_1 that extends wr_2^k by appending rd_1, and where all objects are correct (see Figure 7.1 (a) after deleting the crossed steps).

Partial run pr_2 extends Δpr_1 by appending a complete read rd_2 by r_2 that skips B_3 and B_2 in round one and two respectively (see Figure 7.1 (b)). In pr_2, all objects in block B_2 are malicious, and forge their state to σ_{k-1} before replying to rd_2. By atomicity rd_2 returns 1. Observe that r_2 cannot distinguish pr_2 from some partial run Δpr_2, that extends wr_3^k by appending an incomplete rd_1 and a complete rd_2 and where all objects are correct (Figure 7.1 (b) after deleting the crossed steps).

Partial run pr_3 extends Δpr_2 by appending a complete read rd_3 by r_3 that skips B_4 in round one and B_3 in round two (Figure 7.1 (c)). In pr_3, all objects in block B_3 are malicious, and forge their state to σ_{k-1} before replying to rd_3. By atomicity rd_3 returns 1. Let σ_1^r denote the state of the objects in block B_4 in run pr_3 before replying to rd_2. Observe that r_3 cannot distinguish pr_3 from some partial run Δpr_3, that extends wr_4^k by appending incomplete reads rd_1 and rd_2 and a complete read rd_3 and in which (1) all objects in B_4 are malicious and (2) they forge their state to σ_1^r before replying to rd_2 (Figure 7.1 (c) after deleting the crossed steps). Note that in pr_3, rd_3 completes the second round based on replies only from correct objects, and similarly the first round in Δpr_3. Since r_3 cannot distinguish pr_3 and Δpr_3, it cannot wait for more replies in any of the two runs.

Partial run pr_4 (illustrated in Figure 7.1 (d)) extends Δpr_3 by appending a complete read rd_4 by r_4 that skips B_1 in round one and B_4 in round two. In pr_4, all objects in block B_4 are malicious and forge their state (1) to σ_1^r before replying to rd_2 and (2) to σ_0 before replying to rd_4. By atomicity rd_4 returns 1. Let σ_2^r denote the state of the objects in block B_1 before replying to rd_3. Observe that r_4 cannot distinguish pr_4 from some partial run Δpr_4, that extends wr_1^{k-1} by appending incomplete reads rd_2, rd_3 and a complete read rd_4, and in which (1) all objects in B_1 are malicious and (2) they forge their state to σ_2^r before replying to rd_3 (Figure 7.1 (d) after deleting the

crossed steps). Note that in run pr_4, rd_4 receives second round replies from $n-t$ correct objects. Similarly in Δpr_4, rd_4 receives first round replies from $n-t$ correct objects. Since r_4 cannot distinguish pr_4 and Δpr_4, rd_4 cannot wait for additional replies without violating termination.

After appending rd_4 and constructing Δpr_4 by deleting all steps from pr_4 which are not "visible" to rd_4, we notice that we have erased all steps in column k of write(1) as well as, deleted all steps of rd_1. Thus, we can recycle r_1 by appending rd_1 again and start deleting the steps in column $k-1$.

Starting from Δpr_4 we iteratively define the following partial runs for $1 \leqslant i \leqslant k-1$ and $1 \leqslant j \leqslant 4$ (see Figure 7.1 (d)-(g)). Partial run pr_{4i+j} extends Δpr_{4i+j-1} by appending rd_j. In pr_{4i+j}, all objects in block B_j are malicious and they forge their state (1) to $\sigma^r_{4i+(j-3)}$ before replying to rd_{j-2}[1] and (2) to $\sigma_{((j \mod 4)/j)(k-i-1)}$ before replying to rd_j. Let $\sigma^r_{4i+(j-2)}$ denote the state of the objects in block $B_{(j \mod 4)+1}$ before replying to rd_{j-1}. Observe that r_j cannot distinguish pr_{4i+j} from some partial run Δpr_{4i+j}, that extends $wr^{k-i}_{(j \mod 4)+1}$ by appending incomplete reads rd_{j-2} and rd_{j-1} and a complete read rd_j, and in which (1) all objects in $B_{(j \mod 4)+1}$ are malicious, and (2) they forge their state to $\sigma^r_{4i+(j-2)}$ before replying to rd_{j-1} (Figure 7.1 (d)-(g) after deleting the crossed steps). In run Δpr_{4i+j} and pr_{4i+j}, rd_j receives first and second round replies respectively from $n-t$ correct objects. Since r_j cannot distinguish Δpr_{4i+j} and pr_{4i+j}, rd_j cannot wait for additional replies.

Read rd_4 in Δpr_4 returns 1. Since pr_5 extends Δpr_4 by appending rd_1, by atomicity, rd_1 in pr_5 returns 1. However, as r_1 cannot distinguish pr_5 from Δpr_5, rd_1 in Δpr_5 returns 1. In general, since pr_{4i+j} extends Δpr_{4i+j-1} by appending rd_j (for $1 \leqslant i \leqslant k-1$ and $1 \leqslant j \leqslant 4$), and r_j cannot distinguish pr_{4i+j} from Δpr_{4i+j}, it follows by induction that rd_j in Δpr_{4i+j} returns 1. In particular, rd_3 reads 1 in Δpr_{4k-1}. By our construction, Δpr_{4k-1} extends wr^1_4 and wr^1_4 is indistinguishable from a run in which write(1) is never invoked. Hence, rd_3 returns 1 even if no write is invoked, violating atomicity. □

7.4 The Write Lower Bound

In this section we prove the following proposition.

Proposition 2. *: If $n \leqslant 3t + \lfloor t/t_k \rfloor$ and every read completes in three rounds then no* write *implementation I of a multi-reader atomic register exists that completes in* $\min\{R, \lfloor \log(\lceil (3t_k+1)/2 \rceil) \rfloor\}$ *rounds.*

We first prove the following key lemma. In the effort of making its involved proof easier to follow we first proceed through a careful proof setup

[1]Please note that when we write rd_{j-c}, we always mean $rd_{4-((c-j) \mod 4)}$.

that we found worthwhile. To further help follow the proof, we also visualize runs we use in the proof in Figure 7.2.

Lemma 35. *Let $k \geqslant 1$, $t_{-1} = t_0 = 0$ and $t_k = t_{k-1} + 2t_{k-2} + 1$. There is no implementation I of a k-reader atomic storage with $3t_k + 1$ objects and t_k faults such that the* write *completes in k rounds and the* read *completes in three rounds.*

Preliminaries

Recall that w denotes the writer, r_i for $1 \leqslant i \leqslant k$ denote the readers, and s_i for $1 \leqslant i \leqslant n$ denote the objects. The initial value of the register is $\perp$. In the proof, there is only one write operation write(1) by w that writes value 1. We know from [ACKM06] that the lemma is true for $k = 1$; hence, we assume $k \geqslant 2$. Suppose by contradiction that there is an implementation I that uses at most $3t_k + 1$ objects, such that in every partial run of I every write (resp., read) completes in k (resp. 3) rounds.

We partition the set *objects* into $2k + 2$ distinct blocks, $B_0, \ldots, B_{k+1}$ and $C_1, \ldots, C_k$ such that $|\bigcup_{j=0}^{k+1} B_j| = 2t_k + 1$ and $|\bigcup_{j=1}^{k} C_j| = t_k$. Block B_0 contains a single object. For $1 \leqslant l \leqslant k$, the size of B_l is $t_l - t_{l-2}$ and the size of B_{k+1} is $2t_k + 1 - |\bigcup_{j=0}^{k} B_j| = t_k - t_{k-1}$. For $1 \leqslant l \leqslant k-1$, the size of C_l is $t_{l-1} - t_{l-2}$ and the size of C_k is $t_k - |\bigcup_{j=1}^{k-1} C_j| = t_k - t_{k-2}$. Note here that C_1 is empty. Moreover, we use the abbreviation $BL_{i,j}$ to denote the set $\{BL_i, BL_j\}$, for some $BL \in \{B, C\}$.

We also define three sets of blocks called *superblocks*: the "malicious" superblock $\mathcal{M}_l$, the "parity" superblock $\mathcal{P}_l$ and the "correct" superblock $\mathcal{C}_l$. Superblock $\mathcal{M}_l$ contains all blocks with index at most l, i.e., $\mathcal{M}_l = \{B_0, B_1, \ldots B_l, C_1, \ldots, C_l\}$ for $0 \leqslant l \leqslant k-1$, and $\mathcal{M}_{-1} = \varnothing$. Superblock $\mathcal{P}_l$ contains all blocks B_j with index $j \geqslant l \geqslant 1$ such that j and l have the same parity. More formally, for $1 \leqslant l \leqslant k$, we define $\mathcal{P}_l = \{B_j | l \leqslant j \leqslant k+1 \wedge j \equiv (l \mod 2)\}$. For instance, if k is even then $\mathcal{P}_1 = \{B_1, B_3, \ldots, B_{k-1}, B_{k+1}\}$ and $\mathcal{P}_2 = \{B_2, B_4, \ldots, B_{k-2}, B_k\}$. Finally, superblock $\mathcal{C}_l = \{C_l, \ldots, C_k\}$.

Given the size of the individual blocks, we can determine the cardinality of the union of all elements of a superblock. Namely, if $\mathcal{S} \in \{\mathcal{M}_l, \mathcal{P}_l, \mathcal{C}_l\}$, then we define the union of its elements as $\bigcup \mathcal{S} = \{s \in BL | BL \in \mathcal{S}\}$. Having in mind that $t_k = t_{k-1} + 2t_{k-2} + 1$ (Def.) and $t_{-1} = t_0 = 0$, we have:

$$|\bigcup \mathcal{M}_l| = t_l + 2t_{l-1} + 1 \stackrel{(Def.)}{=} t_{l+1} \qquad \text{for } 0 \leqslant l \leqslant k-1 \tag{7.1}$$

$$|\bigcup \mathcal{P}_l| = t_k - t_{l-2} \quad \text{for } \ 1 \leqslant l \leqslant k+1 \tag{7.2}$$

$$|\bigcup \mathcal{C}_l| = t_k - t_{l-2} \quad \text{for} \;\; 1 \leqslant l \leqslant k \tag{7.3}$$

Block diagrams

Figure 7.2 illustrates the proof for $R = k = 4$. Reader r_i invokes read rd_l, $1 \leqslant l \leqslant k$. In the column corresponding to some round rnd of op we draw a rectangle in a given row, iff round rnd of op does not skip[2] the corresponding block BL. We write "@" in the row of BL iff BL is malicious.

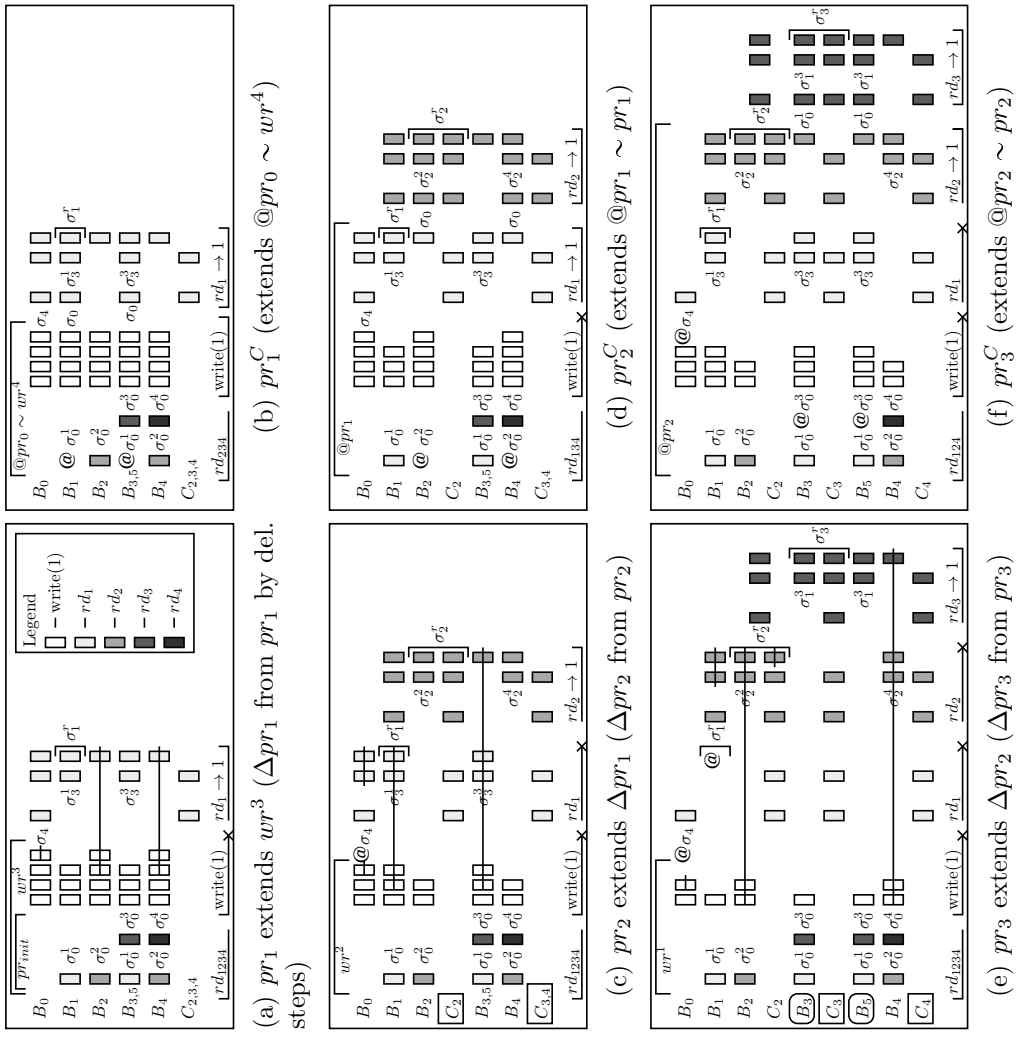


Figure 7.2: (**Part 1**) Instance of the proof with $k = 4$.

[2]The definition of *skipping* extends here from Sec. 7.3.

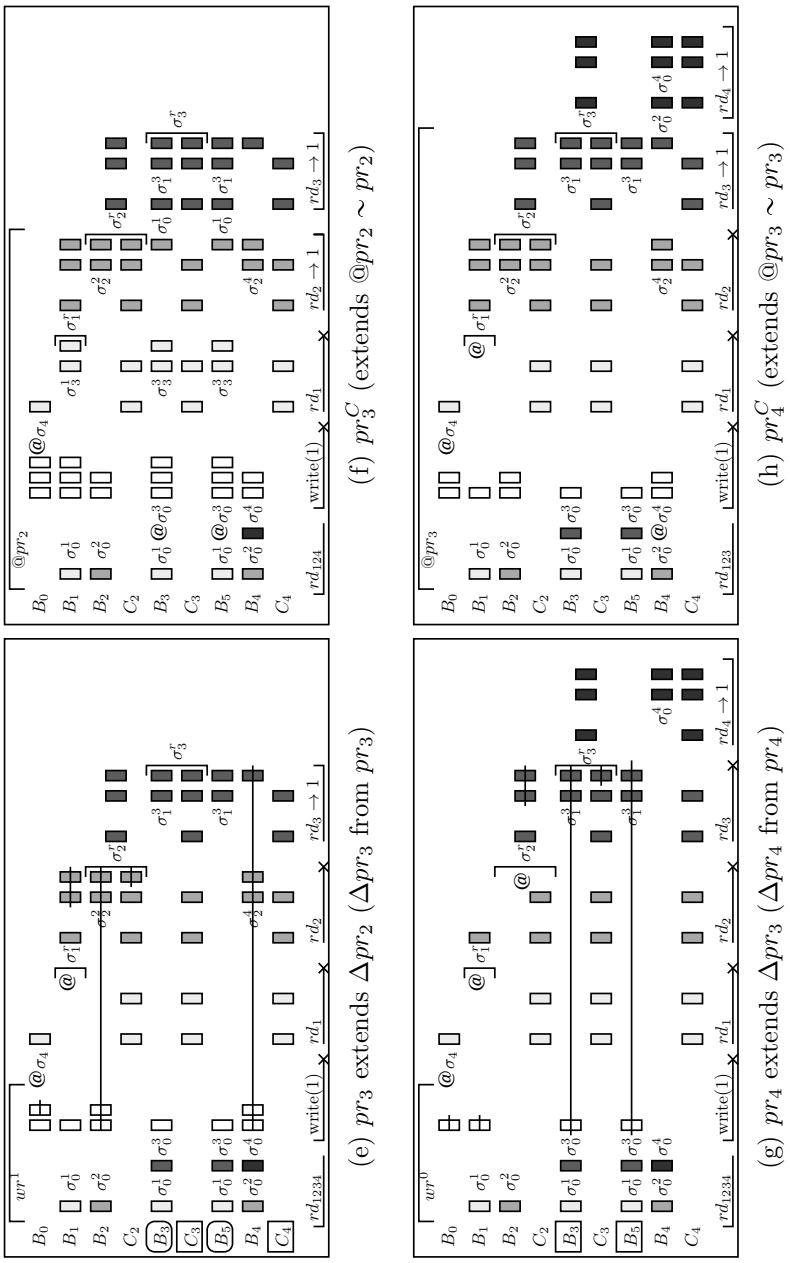


Figure 7.2: (**Part 2**). Instance of the proof with $k = 4$.

Read patterns

We first characterize a *complete* read rd_l for $1 \leqslant l \leqslant k-1$. A complete rd_l skips (a) $\mathcal{M}_{l-2} \cup \mathcal{P}_{l+1}$ in round one and two, and (b) $\mathcal{M}_{l-2} \cup \mathcal{C}_{l+1}$ in round three. Read rd_k skips $\mathcal{M}_{k-2} \cup \mathcal{P}_{k+1}$. Observe that by equations 7.1, 7.2 and 7.3, a read skips exactly t_k objects in each round.

Consider the example in Figure 7.2. Complete reads rd_1, rd_2 and rd_3 skip (respectively): (a) $\{B_{2,4}\}$, $\{B_0\} \cup \{B_{3,5}\}$ and $\{B_{0,1}\} \cup \{B_4\}$ in rounds one and two, and (b) $\{C_{2,3,4}\}$, $\{B_0\} \cup \{C_{3,4}\}$ and $\{B_{0,1}\} \cup \{C_4\}$ in round three. Read rd_4 skips $\{B_{0,1,2}, C_2\} \cup \{B_5\}$.

We further define three types of *incomplete* reads *inc1*, *inc2* and *inc3*, depending on the read's progress. For $1 \leqslant l \leqslant k$, read rd_l is of type *inc1* if the first round is not terminated and skips all blocks *except* $\mathcal{P}_l$. For $1 \leqslant l \leqslant k-1$, read rd_l is of type (a) *inc2* if the first round is terminated, and the second round is not terminated and skips all blocks *except* $\mathcal{C}_l$, and (b) *inc3* if the second round is terminated and the third round is not terminated and skips $\mathcal{M}_{l-2} \cup \mathcal{C}_{l+1} \cup \mathcal{P}_{l+1}$.

Consider our example in Figure 7.2 (c) that illustrates partial run Δpr_2 (after deleting the crossed out steps). Observe that (a) rd_2 is incomplete of type *inc3* (its third round skips $\{B_0\} \cup \{C_{3,4}\} \cup \{B_{3,5}\}$), (b) rd_1 is incomplete of type *inc2* (its second round skips all blocks except $\{C_{2,3,4}\}$) and (b) rd_3 (resp., rd_4) is incomplete of type *inc1*; its first round skips all blocks except $\{B_{3,5}\}$ (resp., $\{B_4\}$).

Towards a contradiction, we construct a partial run of the atomic register implementation I that violates atomicity. More specifically, we exhibit a partial run in which some read returns a value that was never written.

Initialization

Consider a partial run pr_{init} in which (a) all blocks are correct and (b) pr_{init} extends the empty run by appending incomplete reads rd_l by r_l of type *inc1*, for $1 \leqslant l \leqslant k$, one after the other. In pr_{init}, there is no write operation. We refer to the state of each correct block $BL \in \mathcal{P}_l$ after replying to rd_l as σ_0^l. Thus, the state of B_l at the end of pr_{init} corresponds to σ_0^l for $1 \leqslant l \leqslant k$. Further, B_{k+1} is in state σ_0^{k-1}. To see why, note that B_{k-1} and B_{k+1} have the same parity and there are only k reads.

Consider our example Figure 7.2 (a). At the end of pr_{init}, block B_1 (resp., B_2; $B_{3,5}$; B_4) replied to rd_1 (resp., rd_2; rd_1 and rd_3; rd_2 and rd_4); thus, at the end of the run its state is σ_0^1 (resp., σ_0^2; σ_0^3; σ_0^4).

Partial writes

We extend pr_{init} to a partial run wr^k by appending a complete write(1) that completes in k rounds and skips superblock $\mathcal{C}_1$. Moreover, we define a series of partial runs each being a prefix of wr^k. For $1 \leqslant i \leqslant k$, let wr^{k-i} be the partial run which extends pr_{init} by appending an incomplete write(1) such that (i) round 1 to $k-i$ are terminated and (ii) round $k-i+1$ is not

terminated and skips $\mathcal{C}_1$ and all B_j's such that $j > 0$ and i and j have the same parity, i.e., $\mathcal{C}_1 \cup \mathcal{P}_{2-(i \mod 2)}$ (Fig. 7.2 (a) and (c)). We refer to the state of the blocks $B_l \in \mathcal{P}_{2-(i \mod 2)}$ at the end of wr^{k-i} as σ^l_{k-i} for $1 \leqslant l \leqslant k$. If $B_{k+1} \in \mathcal{P}_{2-(i \mod 2)}$, then we refer to its state at the end of wr^{k-i} as σ^{k-1}_{k-i}. Note here that σ^l_{k-i} results from σ^l_0 by appending $k-i$ rounds of the write. When the context is clear, for simplicity we refer to these states using the implicit notation σ^*_{k-1}. Finally, we refer to the state of B_0 at the end of runs wr^k and wr^{k-1} as σ_k.

We refer to our example in Figure 7.2 (a),(c),(e) and (g) for illustrations of the runs wr^3 to wr^0 and the corresponding states. For instance Figure 7.2 (a), illustrates wr^3 as an extension of pr_{init}. The states of the blocks B_0, B_1 and $B_{3,5}$ at the end of wr^3 are σ_4 (4 rounds of write), σ^1_3 and σ^3_3 (3 rounds of write).

Appending Reads

Partial run pr_1 extends wr^{k-1} by appending the missing steps of a complete read rd_1. In pr_1 all objects are correct and thus rd_1 receives replies from $n - t_k$ correct objects. After receiving the third round replies, rd_1 completes and returns value x. We now show that $x = 1$. We define a partial run $@pr_0$, (Fig. 7.2(b)) which is *identical* to wr^k except that in $@pr_0$ (a) no read by r_1 occurs and (b) superblock $\mathcal{P}_1$ is malicious and mimics the occurrence of rd_1 by forging its initial state to σ^1_0. By equation 7.1, the malicious objects in $@pr_0$ amount to t_k. Partial run pr^C_1 (Fig. 7.2(b)) is defined as an extension of $@pr_0$ by appending a complete read rd_1. Read rd_1 cannot distinguish pr^C_1 from pr_1 because $\mathcal{P}_1$, which is malicious in pr^C_1, mimics pr_1. Specifically, $\mathcal{P}_1$ forges its state to σ_0 before replying to rd_1's first round, and then to σ^*_{k-1} before replying to rd_1's second round. In pr^C_1, by atomicity rd_1 returns 1. Since pr^C_1 and pr_1 are indistinguishable to reader r_1, $x = 1$.

Next, we define partial run Δpr_1 obtained from pr_1 by deleting the steps of the read and the write as illustrated in Figure 7.2 (a). More specifically, Δpr_1 extends wr^{k-2} by appending the missing steps of an incomplete read rd_1 of type $inc3$, after which rd_1 crashes. In Δpr_1, $\mathcal{M}_0 = \{B_0\}$ is malicious and forges its state to σ_k before replying to rd_1. Observe that at the end of pr_1 and Δpr_1, every correct block is in the same state, except $\mathcal{P}_2$. We refer to the state of B_1 at the end of Δpr_1 as σ^r_1.

Starting from Δpr_1 we iteratively define the following partial runs for $2 \leqslant l \leqslant k$ (see Fig. 7.2). Partial run pr_l extends Δpr_{l-1} by appending the missing steps of a complete read rd_l. In pr_l, superblock $\mathcal{M}_{l-2}$ is malicious and all other blocks are correct. Since rd_l does not receive any messages from $\mathcal{M}_{l-2}$, it completes only on the basis of replies from correct objects (at least

$n - t_k$ by equation 7.1). At the end of pr_l, rd_l completes and returns value x. To show that $x = 1$, we define a partial run $@pr_{l-1}$ which is identical to pr_{l-1} except that in $@pr_{l-1}$ (a) there is no read by r_l and (b) in addition to $\mathcal{M}_{l-3}$, superblock $\mathcal{P}_l$ is malicious and forges its state to σ_0^l, simulating the occurrence of rd_l as in pr_{l-1}. The count of malicious objects in $@pr_{l-1}$ is exactly t_k. To see why, notice that by equation 7.1 and 7.2 the malicious objects in $@pr_{l-1}$ amount to $|\bigcup \mathcal{P}_l| + |\bigcup \mathcal{M}_{l-3}| = t_k - t_{l-2} + t_{l-2} = t_k$.

Then, partial run pr_l^C extends $@pr_{l-1}$ by appending rd_l. Note that rd_l cannot distinguish pr_l^C from pr_l because superblock $\mathcal{P}_l$, which is malicious in pr_l^C, mimics pr_l. In particular, $\mathcal{P}_l$ forges its state to σ_0 before replying to rd_l's first round and then to σ_{k-l}^* before replying to rd_l's second round. By atomicity, rd_l returns 1 in pr_l^C. Since pr_l^C and pr_l are indistinguishable to reader r_l, $x = 1$.

Next, we define partial run Δpr_l. For $2 \leqslant l < k$, Δpr_l is obtained from pr_l by deleting steps of rd_l, rd_{l-1} and the write (see Fig. 7.2 (c) and (e)). In Δpr_l, superblock $\mathcal{M}_{l-1}$ is malicious, all other block are correct, and blocks $\{B_{l-1}, C_{l-1}\} \in \mathcal{M}_{l-1}$ forge their state to σ_j^r before replying to rd_l.[3] In more detail, Δpr_l extends wr^{k-l-1} by appending the missing steps (a) of incomplete reads $rd_1, \ldots, rd_{l-1}$ of type *inc2*, and (b) of an incomplete rd_l of type *inc3*. B_0 forges its state to σ_k before replying to rd_1 and for $1 \leqslant j \leqslant l-1$, $\{B_j, C_j\}$ forge their state to σ_j^r before replying to rd_{j+1}. Observe that at the end of pr_l and Δpr_l, every correct block is in the same state, except $\mathcal{P}_{l+1}$. We refer to the state of $\{B_l, C_l\}$ at the end of Δpr_1 as σ_l^r.

Finally, partial run Δpr_k is obtained analogously from pr_k, except that in Δpr_k, (a) no write is invoked and (b) read rd_k is complete and skips $\mathcal{M}_{k-2} \cup \mathcal{P}_{k+1}$ (see Fig. 7.2 (g) for $k = 4$). In particular, in Δpr_k, $\mathcal{M}_{k-1}$ is malicious and blocks $\{B_{k-1}, C_{k-1}\} \in \mathcal{M}_{k-1}$ forge their state to σ_{k-1}^r before replying to rd_k. By equation 7.1 the malicious objects amount to $|\bigcup \mathcal{M}_{k-1}| = t_k$. Partial runs pr_k and Δpr_k differ only at $\mathcal{P}_{k+1}$, and rd_k completes without receiving any message from $\mathcal{P}_{k+1}$. Thus, rd_k cannot distinguish Δpr_k from pr_k and returns 1 in Δpr_k, a contradiction, as no write was invoked. □

Lemma 36. *: If $n \leqslant 3t + 1$ and every read completes in three rounds then no* write *implementation I of a multi-reader (MRSW) atomic register exists that completes in* $\min\{R, \lfloor \log(\lceil (3t+1)/2 \rceil) \rfloor\}$ *rounds.*

Proof. Let $k = \min\{R, \lfloor \log(\lceil (3t+1)/2 \rceil) \rfloor\}$, i.e., $R \geqslant k$ and $k \leqslant \lfloor \log(\lceil (3t+1)/2 \rceil) \rfloor$. By Lemma 35, there exists no optimally resilient k-reader atomic register implementation with $t_k = t_{k-1} + 2t_{k-2} + 1$ faulty objects, where

[3]The states are different and are indexed by the object's id, which for simplicity of presentation is made implicit.

the read completes in three rounds and the write completes in k rounds. Observe that this is valid even with $R \geqslant k$ readers and $t \geqslant t_k$ faults. Writing t_k in closed form results in $t_k = \frac{1}{6}(2^{k+2} - (-1)^k - 3)$. Thus, we have that $t \geqslant \frac{1}{6}(2^{k+2} - (-1)^k - 3)$. Solving for k results in $k \leqslant \lfloor \log(\lceil (3t+1)/2 \rceil) \rfloor$. □

Finally, we generalize our result to a resilience of $3t + \lfloor t/t_k \rfloor$ for $t \geqslant t_k$, proving Proposition 2.

Proof. Without loss of generality we can assume that $t \geqslant t_k$ because every implementation is subject to the resilience lower bound of $n \geqslant 3t + 1$. The observation is that if we multiply each of the blocks in the proof of Lemma 35 with a constant c, then the result holds for $n' = cn = 3ct_k + c$ objects and ct_k faults. By carefully choosing $c = t/t_k$, we obtain a lower bound proof for $n' = 3t + \lfloor t/t_k \rfloor$ and t faults. □

7.5 Summary of the Results

In this chapter, we have shown that no multiple-reader single-writer (MRSW) robust atomic storage implementation exists if (a) read operations complete in less than *four* communication round-trips (rounds), and (b) the time complexity of write operations is constant.

However, we observe that a matching implementation can simply be obtained by (a) reusing the MRSW regular storage implementation of [GV06] which features the worst-case time complexity of 2 rounds for both reads and writes, and (b) transforming it to the MRSW atomic implementations using a standard MRSW regular to MRSW atomic transformation technique [Lyn96, AW98]. In short, this transformation employs $R+1$ regular registers, one dedicated to the writer and R additional ones, one per reader, in which a given reader writes back the read value. This yields a sought MRSW atomic implementation in which write operations complete in 2 rounds whereas reads complete in 4 rounds.

Furthermore, in the stronger unauthenticated model enhanced with secret values (see Chapter 6), the regular storage of [GV06] can be replaced in the above transformation with the corresponding time-optimal regular implementation, yielding a 2-round write 3-round read atomic storage, which is optimal in this model. In both models, multi-writer atomic storage can be implemented by applying the standard transformations further [Lyn96, AW98].

In summary, we present two lower bounds. The first is a read lower bound stating that *three* rounds of communication are necessary to read from a MRSW robust atomic storage. The second is a write lower bound, showing that $\Omega(log(t))$ write rounds are necessary to read in three rounds

from such a storage. Our results close a fundamental gap: we show that time-optimal, 2-round write 4-round read (resp. 3-round read in the secret value model) robust atomic storage can be obtained using well-known transformations from regular to atomic storage and existing time-optimal regular storage implementations.

Chapter 8

Conclusion

This thesis investigated the cost in terms of time complexity of implementing consensus and read/write storage, two fundamental abstractions in distributed computing. These primitives are essential for implementing reliable services and data storage in distributed systems. The thesis' results consist of improved consensus algorithms tolerating crash failures, and enhanced distributed storage implementations resilient to Byzantine failures. The presented algorithms provide optimal resilience and/or optimal complexity. We now briefly summarize our contributions and outline a few open directions for future exploration.

One-step Consensus with Zero-Degradation One-step decision and zero-degradation are key efficiency properties of indulgent consensus implementations. They express the ability to reach consensus in a single step when all proposals are equal, yet to gracefully degrade to two steps in stable executions, which is optimal. We investigated if these properties are incompatible and showed that they cannot be both satisfied using the Ω failure detector. Then, we proposed two approaches and corresponding protocols, to circumvent the impossibility. The first approach relaxes one-step decision to hold only in stable runs. The second approach assumes the strictly stronger $\diamond\mathcal{P}$ failure detector. Further we have devised a consensus-based atomic broadcast implementation, that using our consensus protocols is optimal, requiring two communication steps in the absence of collisions, and three in the common case. We have compared our algorithms with Paxos both analytically and experimentally. The results of the experiments confirm the efficiency of our protocols for low to medium load, i.e., when only few collisions occur. However, with increasing load, Paxos performs better. This is justified by the high message overhead generated by our protocols relative to Paxos.

We have identified two possible open problems mainly of theoretical inter-

est. One would be to generalize our lower bound also for values of $f < \lfloor n/4 \rfloor$. Another open question is if $\diamond\mathcal{P}$ is also the weakest failure detector that enables one-step decision and zero-degradation.

Generalized Consensus and Hybrid Paxos We have devised Hybrid Paxos, a generalized consensus protocol resilient to crash failures, matching all known lower bounds on latency, resilience and messages. The core idea of Hybrid Paxos is to enhance Classic Paxos with fast learning capabilities, yet without overloading the leader process, which easily becomes the bottleneck. Hybrid Paxos is to our knowledge the first generalized consensus protocol that attains the optimal latency (a) of two message delays in the absence of collisions caused by interfering commands and (b) of three message delays in the common case. Moreover, our implementation exhibits optimal resilience and optimal messages. We have shown that generalized consensus is a practical approach to replication in a WAN by means of a simple banking application. Our experimental results demonstrate that HP can outperform state of the art protocols.

With Generalized Paxos and Hybrid Paxos as well, fast learning is possible only if a fast quorum exists consisting of $\lceil \frac{n+f+1}{2} \rceil$ acceptors, which is strictly larger than a majority quorum. An extension of Hybrid Paxos, in which fast quorums are *predefined*, in the flavor of [GR04], would allow fast learning with quorums consisting of as few as $\lceil \frac{n+1}{2} \rceil$ acceptors. In practice, this would be a viable alternative to our approach, albeit under the additional assumption that the set of responsive acceptors does not "jump around" too fast, for instance as required by [LM04].

Robust Amnesic Storage We have presented amnesic algorithms that robustly implement a shared register from storage objects prone to Byzantine failures. We have shown that two rounds of communication with the base objects are sufficient for reading from amnesic robust storage with optimal resilience. Our result is optimal, matching the two-round lower bound proved in [GV06]. Also we have devised the first *fast* implementation of robust and amnesic storage, i.e., one in which both reads and writes complete in one communication round. Our algorithm from $4t + 1$ base objects is optimal: with $4t$ base objects, both the read and the write operations require at least two communication rounds as previously shown in [ACKM06, GV06].

Our results provide a first (theoretical) step towards space efficient robust storage algorithms. For the deployment in a real distributed storage environment, many problems still need to be addressed. For instance, in the optimally-resilient case, the write operation needs *three* rounds of communi-

cation with the servers, whereas the optimal number is *two* rounds.

Secondly, our algorithms implement only a single-reader register. A straightforward construction of a multiple-reader register can be realized using m copies of the single-reader register, one for each reader. In a distributed setting, the writer accesses all copies in parallel, whereas the reader accesses a single copy. Although correct, this construction is highly inefficient, because the writer has to store each value m times. Here, the challenge is to devise protocols which are more efficient in terms of bandwidth and space usage.

To further improve the throughput and the memory consumption at the servers, our algorithms could be combined with the powerful approach of erasure coding. Instead of storing a complete copy of the data, each server holds a share, such that the original data can be reconstructed from enough servers' portions. Existing practical distributed storage systems utilizing erasure coding are either not amnesic or they are not robust. Specifically, in [HGR07], read operations are guaranteed to terminate only in the absence of contention.

Some of the prior amnesic (but not robust) register implementations assume that the readers cannot modify the server state, e.g. [ACKM06]. This assumption results in implementations that possess several properties that could be valuable in practice, for instance the ability to tolerate any number of malicious readers while using only $\mathcal{O}(1)$ memory at the servers. We are not aware of any robust implementation supporting that as well, and in fact, our algorithms are not an exception. We leave as an open problem the question whether robust and amnesic register implementations exist, that would support any number of readers while using only $\mathcal{O}(1)$ memory at the servers.

Robust Storage with Secret Tokens We have devised robust algorithms that close the existing complexity gap between unauthenticated storage and storage relying on self-verifying data.

The algorithms presented effectively circumvent lower bounds established for unauthenticated data storage by means of secret tokens. The first algorithm supports unbounded readers and two-rounded reads. The second algorithm features fast reads. Even if the secrecy assumption of the token is violated, both algorithms are gracefully degrading. The first algorithm fully preserves regularity and the second algorithm never blocks and never returns a forged value. Moreover, the probability that the properties of our algorithms are violated is negligibly small if the token space is large enough. The algorithms are secure against a computationally unbounded adversary because tokens are random and thus cannot be computed.

Our algorithms are non-amnesic, requiring base objects to store all the values they receive from the writer. If readers do not write, storing less values is an open problem [CGK07]. Although some very practical storage systems [GWGR04] rely on the same assumption, this might raise issues of storage exhaustion and needs careful garbage collection.

The second algorithm supporting fast reads relies on readers writing into the base objects. Thus, an arbitration mechanism between reader and writer as shown in Chapter 5 could be employed to obtain an amnesic algorithm. However, we observe that the arbitration relies on having the readers increment a safe counter, which implies writing to at least a safe register. However, we know that writing to a safe register with optimal resilience takes at least two rounds [ACKM06]. Hence, it remains an interesting open problem if a robust amnesic algorithm supporting fast read operations exists at all.

Robust Atomic Storage Complexity We have shown two lower bounds on the time complexity of robust atomic storage, where by robust we mean algorithms that wait-free implement a storage tolerating the largest number of Byzantine base object failures (i.e. optimal resilience).

The first result is a read lower bound stating that no multiple-reader robust atomic storage implementation exists if read operations complete in two communication rounds. The second is a write lower bound, showing that $\Omega(log(t))$ write rounds are necessary to read in three rounds from such a storage. Our results close a fundamental gap: we show that time-optimal, 2-round write 4-round read (resp. 3-round read in the secret value model (Chapter 6)) robust atomic storage can be obtained using well-known transformations from regular to atomic storage and existing time-optimal regular storage implementations.

We observe that a matching implementation can simply be obtained by (a) reusing the MRSW regular storage implementation of [GV06] which features the worst-case time complexity of 2 rounds for both reads and writes, and (b) transforming it to the MRSW atomic implementation using a standard regular to atomic transformation technique [Lyn96, AW98]. This yields the sought MRSW atomic implementation in which write operations complete in 2 rounds whereas reads complete in 4 rounds.

Furthermore, in the stronger authentication model that allows for secret values (Chapter 6), the regular storage described in [GV06] can be replaced in the above transformation with the corresponding regular implementation from Chapter 6. The result is a 2-round write 3-round read atomic storage, which is optimal in this model. In both models, multi-writer atomic storage can be implemented by applying the standard transformations further

[Lyn96, AW98].

One possible avenue for future research would be to devise an algorithm that matches our write lower bound, i.e., an algorithm that supports three-rounded read operations with $\mathcal{O}(\log(t))$ write rounds. Note that the practical relevance of such an algorithm would be limited, unless the number of write rounds can be described as a function of the bound on *malicious* faults. Then, such an algorithm could be applicable to systems designed to support a small fraction of malicious faults, yet a large number of benign faults.

From a practical viewpoint, it would be interesting to explore ways to circumvent the read lower bound (stating that three read rounds are necessary), by ways of additional assumptions. We believe that hiding a secret token from the adversary, e.g., by employing a hash function, and then revealing the secret later, would help atomic reads to complete in two rounds in the worst case. This would be in line with the complexity of atomic storage relying on self-verifying data and that of crash-tolerant atomic storage.

Appendix A

Computing Digests of Large Histories

In this section we show how to efficiently compute the digest of a large history. We describe an algorithm based on incremental hashing that leverages knowledge about the application, illustrated in Figure A.1. The application considered is the banking example described in Section 4.5.1.

Recall that, our banking system provides two command types: *deposit* and *withdraw*. Commands of type *withdraw* are not commutable with any other command and *deposit* commands commute with each other. The description of HP in section 4.4 abstracts away the details of the command history data structure to simplify the presentation. The command history type maintains the following attributes: a *type* field to distinguish between a *classic* and *fast* history, a *cache* field which stores for each command the digest of the history up to that command and the field *digest*, used to store the history digest. The *digest* field consists of an incremental hash value of the history (described below). The history type has the additional application specific field *Deposit*, which is an ordered queue containing *deposit* commands.

Since commands of type *withdraw* are not commutable with any other command, for every withdraw command w in a command history h we can distinguish between commands which are ordered before or after w in h. To compute incremental history digests for history h using `historyDigest` the idea is the following: the digest of the history up to the last *withdraw* command w appended to history h is maintained in $h.digest$. For any deposit command d appended after w and before the next *withdraw* command, the history up to d is completely determined by the history up to w. This is true because it subsumes all operations that must be ordered before d. Thus, the digest of d can be computed incrementally as $H(h.digest \circ d.id)$

(line 7), where $\circ$ is the concatenation operator and $d.id$ is the command identifier. When the next withdraw command w' is appended, $h.digest$ is updated. The new digest of h up to w' depends on $h.digest$ (which is the digest of h up to w) and all deposit commands which are appended to h between w and w'. Such commands $d_1, d_2, \ldots$ are collected in the ordered queue *Deposit* (Whenever a digest for a deposit command is computed, the command is added to *Deposit* in line 6). Therefore, $h.digest$ is updated with $H(h.digest \circ d_1.id \circ d_2.id \circ \ldots \circ w'.id)$ (line 10) and *Deposit* (lines 8–9) is cleared.

Additionally, when a digest for command c is computed, c and the corresponding digest are added to $h.cache$ (line 12). When the `historyDigest` function is called for a *classic* history h and an entry for c is contained in $h.cache$ (line 3), the corresponding digest is directly returned without any computation (line 4).

```
1  tmp type string, initially empty
2  digest type byte array, initially empty
3  if
   history.cache.contains(cmd.id) ∧ history.type = CLASSIC
   then
4      return history.cache.get(cmd.id); /* return digest
          from cache                                      */
   /* construct digest incrementally                     */
5  if cmd.type = DEPOSIT then
6      history.Deposit.push(cmd.id)
7      digest ← H(history.digest ∘ cmd.id)
   else
       /* WITHDRAW command                               */
8      while history.Deposit ≠ ∅ do
9          tmp ← tmp ∘ history.Deposit.pop()
10     digest ← H(history.digest ∘ tmp ∘ cmd.id)
11     history.digest ← digest
12 history.cache.put(cmd.id, digest) /* cache history digest
       */
13 return digest
```

Figure A.1: **Function** `historyDigest`(*cmd*, *history*)

Appendix B

Read Lower Bound (The Hybrid Model)

In this section we prove the result of Section 7.3 correct in the hybrid model of [TP88], where at most b out of t objects may be malicious and the rest of $t - b$ are simple crash failures. Here we assume $b > 0$.

Proposition 3. *: If $n \leqslant 2t + 2b$ and $R > 3$, then no* read *implementation I of a multi-reader (MRSW) atomic register exists that completes in two rounds.*

Preliminaries. In the proof w denotes the writer, r_i for $1 \leqslant i \leqslant 4$ denote the readers, and s_i for $1 \leqslant i \leqslant S$ denote the objects. Suppose by contradiction that $R = 4$ and there is an atomic register implementation I that uses at most $2t + 2b$ objects, such that in every partial run of I every *read* operation completes in two communication rounds.

We partition the set *objects* into six disjoint subsets (which we call *blocks*), denoted B_i for $1 \leqslant i \leqslant 4$ each of size at most $b \geqslant 1$, and C_j for $1 \leqslant j \leqslant 2$ each of size $t - b$. Since, $|B_i \cup C_j| \leqslant t$ all objects in $B_i \cup C_j$ may fail together. Throughout the proof, the blocks B_i fail Byzantine and C_j fail by crashing.

Without loss of generality we can assume that $S \geqslant 2t + 2$ because every implementation I must conform to the resilience lower bound of $S \geqslant 2t + b + 1$ [MAD02] (recall that we assume $b \geqslant 1$). Hence, we can assume that the blocks B_i contain at least one object. If $b = t$, then the blocks C_1 and C_2 are empty. We refer to the initial state of every correct block B_j as σ_0^j. For simplicity we simply write σ_0, where the block name is implicit.

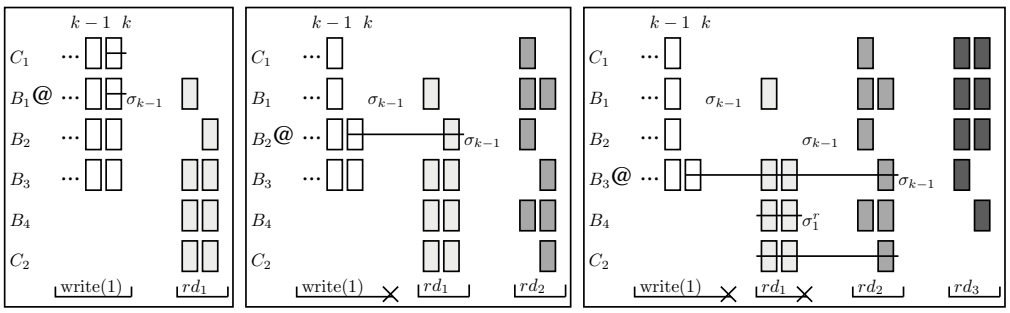
We say that a round *rnd* of an operation *op skips* a set of blocks BS in a partial run, (where $BS \subseteq \{B_1, \ldots, B_4, C_1, C_2\}$), if (1) no object in any block $BL \in BS$ receives any message in round *rnd* from *op* in that partial

run, (2) all other objects receive all messages in round *rnd* from *op* and reply to the messages (3) in case round *rnd* is terminated, the invoking client has received all these reply messages or, in case *rnd* is not terminated, all these reply messages are in transit. We say that an operation *op* skips a set of blocks BS in a partial run if every round of *op* skips BS.

To show a contradiction, we construct a partial run of the implementation I that violates atomicity: a partial run of I in which no value is ever written and some read returns 1.

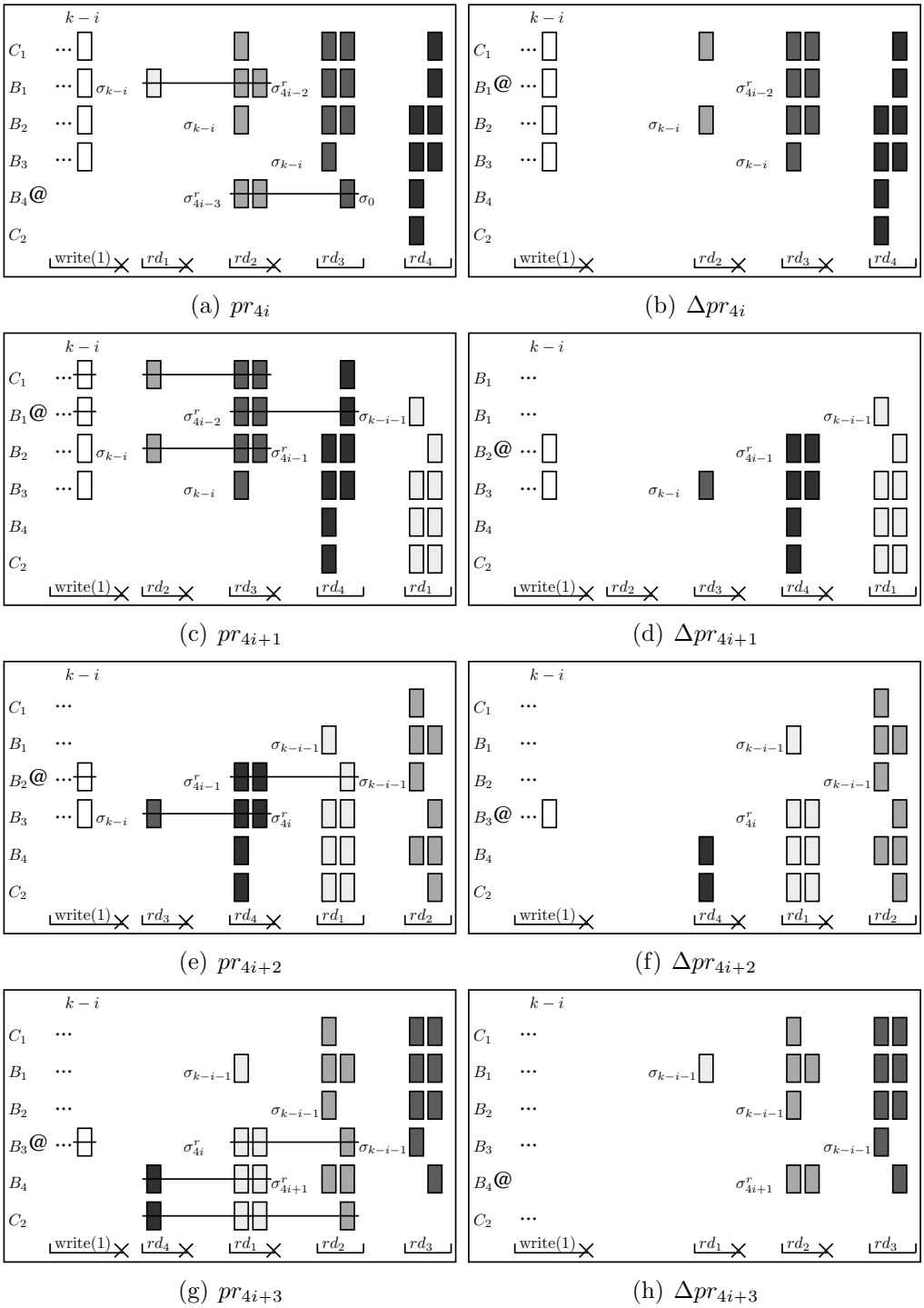
Partial writes.

The initial value of the register is $\perp$. Throughout the proof there is only one write operation write(1) by w that writes value 1. Consider a partial run wr in which w completes write(1) on the register. Let k be the number of rounds invoked by w in wr. We denote the state of every correct block B_j after B_j has replied to the messages of the write during round 1 to i where $1 \leqslant i \leqslant k$, as σ_i^j. For simplicity we refer to σ_i^j as σ_i. The write operation skips blocks $\{B_4, C_2\}$. We define a series of partial runs containing an incomplete write(1) invocation, each being a prefix of wr. For $1 \leqslant i \leqslant k$, we define wr_1^i as the partial run in which (1) rounds 1 to i skip $\{B_4, C_2\}$, (2) rounds 1 to $i-1$ are terminated and round i is not terminated, and (3) all objects are correct. For $1 \leqslant i \leqslant k$ and $2 \leqslant j \leqslant 4$, we define wr_j^i as the partial run in which (1) rounds 1 to $i-1$ are terminated and skip $\{B_4, C_2\}$, (2) round i is not terminated and skips blocks $\{B_1, \ldots, B_4, C_1, C_2\} \backslash \{B_l \mid j \leqslant l \leqslant 3\}$. and (3) all objects are correct. We make two simple observations: (1) partial run wr_1^k differs from wr only at w and (2) partial run wr_4^1 differs from a run in which write(1) is never invoked only at w.



(a) $pr_1 \sim \Delta pr_1$ (b) $pr_2 \sim \Delta pr_2$ (c) $pr_3 \sim \Delta pr_3$

Figure B.1: Runs used in the proof of Proposition 3 (Setup)



(a) pr_{4i} (b) Δpr_{4i}

(c) pr_{4i+1} (d) Δpr_{4i+1}

(e) pr_{4i+2} (f) Δpr_{4i+2}

(g) pr_{4i+3} (h) Δpr_{4i+3}

Figure B.2: Runs used in the proof of Proposition 3

Block diagrams.

We illustrate the proof in Figures B.1 and B.2. We depict a round rnd of an operation op through a set of rectangles arranged in a single column. In the column corresponding to some round rnd of op we draw a rectangle in a given row, if all objects in the corresponding block BL have received the message from the client in round rnd of op and have sent reply messages, i.e., if round rnd of op does not skip BL. We write "@" in the row corresponding to BL iff BL is malicious.

Appending reads.

Partial run pr_1 extends wr by appending a complete read rd_1 by r_1 that skips $\{B_2, C_1\}$ in round one and $\{B_1, C_1\}$ in round two (see Figure B.1 (a)). In pr_1, all objects in block B_1 are malicious, and forge their state to σ_{k-1} before replying to rd_1. By atomicity rd_1 returns 1. Observe that r_1 cannot distinguish pr_1 from some partial Δpr_1 that extends wr_2^k by appending rd_1, and where all objects are correct.

Partial run pr_2 extends Δpr_1 by appending a complete read rd_2 by r_2 that skips $\{B_3, C_2\}$ in round one and $\{B_2, C_1\}$ in round two (see Figure B.1 (b)). In pr_2, all objects in block B_2 are malicious, and forge their state to σ_{k-1} before replying to rd_2. By atomicity rd_2 returns 1. Observe that r_2 cannot distinguish pr_2 from some partial run Δpr_2, that extends wr_3^k by appending an incomplete rd_1 and a complete rd_2 and where all objects are correct (as illustrated in Figure B.1 (b) after deleting the crossed steps).

Partial run pr_3 extends Δpr_2 by appending a complete read rd_3 by r_3 that skips $\{B_4, C_2\}$ in round one and $\{B_3, C_2\}$ in round two. In pr_3, all objects in block B_3 are malicious, and forge their state to σ_{k-1} before replying to rd_3. By atomicity rd_3 returns 1. Let σ_1^r denote the state of the objects in block B_4 in run pr_3 before replying to rd_2. Observe that r_3 cannot distinguish pr_3 from some partial run Δpr_3, that extends wr_4^k by appending incomplete reads rd_1 and rd_2 and a complete read rd_3 and in which (1) all objects in B_4 are malicious and (2) they forge their state to σ_1^r before replying to rd_2 (as shown in Figure B.1 (c) after deleting the crossed steps). Note that since r_3 cannot distinguish pr_3 and Δpr_3, it cannot wait for more replies in order to complete. More precisely, rd_3 completes in Δpr_3 (resp. pr_3) the first (resp. second) round based on replies only from correct objects (at least $S - t$).

Partial run pr_4 (illustrated in Figure B.2 (a)) extends Δpr_3 by appending a complete read rd_4 by r_4 that skips $\{B_1, C_1\}$ in round one and $\{B_4, C_2\}$ in round two. In pr_4, all objects in block B_4 are malicious and forge their state (1) to σ_1^r before replying to rd_2 and (2) to σ_0 before replying to rd_4.

By atomicity rd_4 returns 1. Let σ_2^r denote the state of the objects in block B_1 before replying to rd_3. Observe that r_4 cannot distinguish pr_4 from some partial run Δpr_4, that extends wr_1^{k-1} by appending incomplete reads rd_2, rd_3 and a complete read rd_4, and in which (1) all objects in B_1 are malicious and (2) they forge their state to σ_2^r before replying to rd_3 (see Figure B.2 (b)). Since r_4 cannot distinguish pr_4 and Δpr_4, rd_4 cannot wait for additional replies in order to complete, without violating termination. To see why, notice that in run Δpr_4 (resp pr_4), rd_4 receives first (resp. second) round replies from $S - t$ correct objects.

After appending rd_4 and constructing Δpr_4 by deleting all steps from pr_4 which are not "visible" to rd_4, we notice that we have erased all steps in column k of write(1) as well as, deleted all steps of rd_1. Thus, we can recycle r_1 by appending rd_1 again and start deleting the steps in column $k-1$.

Starting from Δpr_4 we iteratively define the following partial runs for $1 \leqslant i \leqslant k-1$ and $1 \leqslant j \leqslant 4$ (depicted in Figure B.2 (a)-(h)). Partial run pr_{4i+j} extends Δpr_{4i+j-1} by appending rd_j. In pr_{4i+j}, all objects in block B_j are malicious and they forge their state (1) to $\sigma^r_{4i+(j-3)}$ before replying to rd_{j-2}[1] and (2) to $\sigma_{((j \mod 4)/j)(k-i-1)}$ before replying to rd_j. Let $\sigma^r_{4i+(j-2)}$ denote the state of the objects in block $B_{(j \mod 4)+1}$ before replying to rd_{j-1}. Observe that r_j cannot distinguish pr_{4i+j} from some partial run Δpr_{4i+j}, that extends $wr^{k-i}_{(j \mod 4)+1}$ by appending incomplete reads rd_{j-2} and rd_{j-1} and a complete read rd_j, and in which (1) all objects in $B_{(j \mod 4)+1}$ are malicious, and (2) they forge their state to $\sigma^r_{4i+(j-2)}$ before replying to rd_{j-1} (see Figure B.2 (b),(d),(f),(h)). Since r_j cannot distinguish Δpr_{4i+j} and pr_{4i+j}, rd_j cannot wait for additional replies, because in run Δpr_{4i+j} and pr_{4i+j}, rd_j receives first and second round replies respectively from $S - t$ correct objects.

Read rd_4 in Δpr_4 returns 1. Since pr_5 extends Δpr_4 by appending rd_1, by atomicity, rd_1 in pr_5 returns 1. However, as r_1 cannot distinguish pr_5 from Δpr_5, rd_1 in Δpr_5 returns 1. In general, since pr_{4i+j} extends Δpr_{4i+j-1} by appending rd_j (for $1 \leqslant i \leqslant k-1$ and $1 \leqslant j \leqslant 4$), and r_j cannot distinguish pr_{4i+j} from Δpr_{4i+j}, it follows by induction that rd_j in Δpr_{4i+j} returns 1. In particular, rd_3 reads 1 in Δpr_{4k-1}. By our construction, Δpr_{4k-1} extends wr_4^1 and wr_4^1 is indistinguishable from a run in which write(1) is never invoked. Hence, rd_3 returns 1 even if no write is invoked, violating atomicity. □

[1]Please note that when we write rd_{j-c}, we always mean $rd_{4-((c-j) \mod 4)}$.

List of Figures

List of Tables

Bibliography

[AAB07] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In *Proceedings of the 21st International Symposium on Distributed Computing*, pages 7–19, September 2007.

[ABD95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[ACKM06] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[ACKM07] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Wait-free regular storage from byzantine components. *Inf. Process. Lett.*, 101(2), 2007.

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. of DISC*, pages 231–245, 1998.

[ACT00] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure Detection and Consensus in the Crash-Recovery Model. *Distributed Computing*, 13, 2000.

[AEMCC+05] Michael Abd-El-Malek, William V. Courtright, II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: versatile cluster-based storage. In *FAST'05:*

Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[ASV06] Marcos K. Aguilera, Susan Spence, and Alistair Veitch. Olive: distributed point-in-time branching storage for real systems. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[AW98] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[AWS] AWS Simple Storage Service. http://aws.amazon.com/s3/.

[BBC+04] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. of NSDI*, pages 253–266, 2004. *et al.*

[BCBT96] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proc. of WDAG*, pages 105–122, 1996.

[BD04] Rida A. Bazzi and Yin Ding. Non-skipping timestamps for byzantine data storage systems. In *DISC*, pages 405–419, 2004.

[BGMR01] Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in One Communication Step. In *PACT*, 2001.

[BO83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC*, pages 27–30, 1983.

[CDH+06] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *NSDI'06: Proceedings of the*

3rd conference on Networked Systems Design & Implementation, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.

[CF98] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. In *Proc. of FTCS*, 1998.

[CGK07] Gregory Chockler, Rachid Guerraoui, and Idit Keidar. Amnesic Distributed Storage. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC'07)*, 2007.

[CGKV09] Gregory Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolic. Reliable distributed storage. *IEEE Computer*, 42(4):60–67, 2009.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *JACM*, 43, 1996.

[Chu98] Francis Chu. Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 67, 1998.

[CKS09] Christian Cachin, Idit Keidar, and Alexander Shraer. Trusting the cloud. *SIGACT News*, 40(2):81–86, 2009.

[CM05] Gregory Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1):73–84, 2005.

[CMP06] Lásaro J. Camargos, Edmundo R. M. Madeira, and Fernando Pedone. Optimal and practical wab-based consensus algorithms. In *Euro-Par*, pages 549–558, 2006.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, (2):225–267, 1996.

[CT06] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 115–124, Washington, DC, USA, 2006. IEEE Computer Society.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[DG02] P. Dutta and R. Guerraoui. Fast indulgent consensus with zero degradation. In *Proc. of EDCC*, pages 191–208, 2002.

[DG05] Partha Dutta and Rachid Guerraoui. The inherent price of indulgence. *Distrib. Comput.*, 18(1):85–98, 2005.

[DGK07] Partha Dutta, Rachid Guerraoui, and Idit Keidar. The Overhead of Consensus Failure Recovery. *Distributed Computing*, 19(5-6), 2007.

[DGLC04] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd annual ACM symposium on Principles of distributed computing*, pages 236–245, July 2004.

[DGLV05] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. How Fast can a Distributed Atomic Read be? Technical Report LPD-REPORT-2005-001, 2005.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[EGM+09] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, pages 240–254, 2009.

[FL03] Rui Fan and Nancy Lynch. Efficient replication of large data objects. In *Proceedings of the 17th International Symposium on Distributed Computing*, pages 75–91, October 2003.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):372–382, Apr 1985.

[Gea] Geant2. Pan-european backbone network. Website. `http://www.geant2.net`.

[GGK07] Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the message complexity of indulgent consensus. In *DISC*, pages 283–297, 2007.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[GLV06] Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Lucky read/write access to robust atomic storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 125–136, 2006.

[GNS09] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel Distrib. Comput.*, 69(1):62–79, 2009.

[GR04] Rachid Guerraoui and Michel Raynal. The Information Structure of Indulgent Consensus. *IEEE Trans. Comput.*, 53(4), 2004.

[Gra78] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[Gue00] Rachid Guerraoui. Indulgent algorithms (preliminary version). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 289–297, New York, NY, USA, 2000. ACM.

[GV06] Rachid Guerraoui and Marko Vukolić. How fast can a very robust read be? In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 248–257, New York, NY, USA, 2006. ACM.

[GV07] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 119–128, 2007.

[GWGR04] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 135, Washington, DC, USA, 2004. IEEE Computer Society.

[Hem05] Stephen Hemminger. Network emulation with netem. In *Proc. of LCA*, 2005.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HGR07] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead byzantine fault-tolerant storage. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 73–86, New York, NY, USA, 2007. ACM.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522, Washington, DC, USA, 2003. IEEE Computer Society.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[JCT98] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, 1998.

[KR01] I. Keidar and S. Rajsbaum. On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial. *SIGACT News*, 32(2):45–63, June 2001.

[KS06] I. Keidar and A. Shraer. Timeliness, failure-detectors, and consensus performance. In *Proc. of PODC*, pages 169–178, 2006.

[Lam78] L. Lamport. Time, Clocks and Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam86] Leslie Lamport. On interprocess communication. part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[Lam98] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam03] Leslie Lamport. Lower bounds for asynchronous consensus. In *In Proc. FUDICO*, pages 22–23. Springer, 2003.

[Lam05] Leslie Lamport. Generalized consensus and paxos. In *MSR-TR-2005-33*, 2005.

[Lam06a] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[Lam06b] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2), 2006.

[LM04] Leslie Lamport and Mike Massa. Cheap paxos. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 307, Washington, DC, USA, 2004. IEEE Computer Society.

[LR89] Nancy A. Lynch and Mark R.Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LR06] Barbara Liskov and Rodrigo Rodrigues. Tolerating byzantine faulty clients in a quorum system. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 34, Washington, DC, USA, 2006. IEEE Computer Society.

[LS02] Nancy A. Lynch and Alexander A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 173–190, London, UK, 2002. Springer-Verlag.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MAD02] J-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine Storage. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC 2002), LNCS 2508*, pages 311–325, 2002.

[MJM08] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *OSDI*, 2008.

[MR98] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

[MR00] Achour Mostéfaoui and Michel Raynal. Low Cost Consensus-based Atomic Broadcast. In *PRDC*, 2000.

[MR01] Achour Mostefaoui and Michel Raynal. Leader-based consensus. *Prallel Processing Letters*, 11(1):95–107, 2001.

[MRR03] Achour Mostefaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 50(6):922–954, 2003.

[PS02] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2), 2002.

[PS03] Fernando Pedone and André Schiper. Optimistic Atomic Broadcast: A Pragmatic Viewpoint. *TCS*, 291, 2003.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching Agreements in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PSUC02] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving Agreement Problems with Weak Ordering Oracles. In *EDCC*, 2002.

[RC05] H. V. Ramasamy and C. Cachin. Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast. In *Proceedings of the 9th International Conference On Principles Of Distributed Systems (OPODIS-2005), LNCS 3974*, pages 88–102, December 2005.

[Sch90] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec 1990.

[Sch97] André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput.*, 10(3):149–157, 1997.

[SFV+04] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.*, 38(5):48–58, 2004.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 231–244, Berkeley, CA, USA, 2002. USENIX Association.

[SPMO02] A. Sousa, J. Pereira, F. Moura, and R. Oliviera. Optimistic total order in wide area networks. In *Proc. of SRDS*, pages 190–199, 2002.

[TP88] Philip M. Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Symposium on Reliable Distributed Systems*, pages 93–100, 1988.

[Tsu92] G. Tsudik. Message Authentication with One-Way Hash Functions. *ACM Computer Communications Review*, 22(5):29–38, 1992.

[UDS01] Péter Urbán, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *ICOIN*, 2001.

[vRS04] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting*

Systems Design & Implementation, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[WLS+03] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of OSDI*, pages 255–270, 2003.

[Zie05] P. Zielinski. Optimistic generic broadcast. In *Proc. of DISC*, pages 369–383, 2005.

Curriculum Vitae

Dan Dobre was born on the 24th November 1977 in Lugoj, Romania. He completed the Natural Scientific Friedrich-Koenig Highschool in Würzburg, Germany in summer 1997.

After the civilian service, in late fall 1998, Dan started his computer science studies at TU Darmstadt, Germany. He graduated with distinction and obtained his Masters (Dipl.-Inf) in summer 2004.

Late fall 2004, Dan joined the DEEDS group at TU Darmstadt and started his doctoral studies under the supervision of Prof. Neeraj Suri. Dan has won the highly competitive Microsoft Research PhD fellowship award, including a complete funding of his research for three years (2005-2008). During this period, besides his work as researcher, Dan has served as teaching assistant and as reviewer for several conferences and journals.