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.

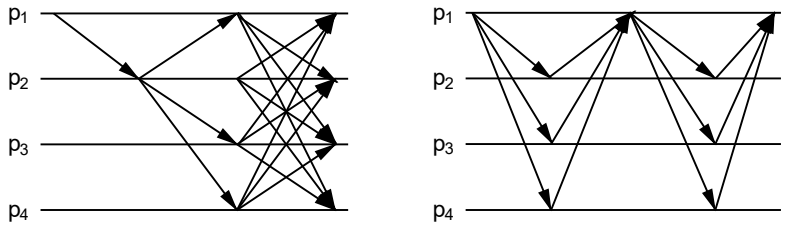
1.1.3 On Time Complexity and Related Metrics

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., $2\mathsf{f}+1$ 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., $2\mathsf{t} + 1$ 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 $3\mathsf{t}+1$ base objects (optimal resilience) [MAD02]. To see why, note that any two quorums must overlap in $\mathsf{t}+1$ 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 $\mathsf{t}+1$ 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 $\diamond\mathsf{P}$, 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 $\diamond\mathsf{P}$ 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 $2\mathsf{f} + 1$ 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 $4\mathsf{t} + 1$ 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 $3\mathsf{t} + 1$ 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 $\mathsf{t} + 1$ 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 $\mathsf{t} + 1$ 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 $\mathsf{t} + 1$ 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 ($\mathsf{k} = \Omega(\mathsf{log}(\mathsf{t}))$).

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 $\mathsf{A_p}$ 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 $\langle \mathsf{p}, \mathsf{M} \rangle$ (M might be $\varnothing$). In step $\mathsf{sp} = \langle \mathsf{p}, \mathsf{M} \rangle$, 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 $\mathsf{st_p}$ to $\mathsf{A_p}$, which outputs a new state $\mathsf{st'_p}$ and a set of messages to be sent, and then p adopts $\mathsf{st'_p}$ 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 $\diamond P$. Both eventually provide consistent and correct information about the state of processes, i.e., crashed or non-crashed. While $\diamond P$ eventually outputs exactly the crashed processes, Ω eventually outputs a single correct *leader* process. Ω is strictly weaker than $\diamond P$ and it is the weakest failure detector to solve consensus [CHT96, Chu98]. More formally, $\diamond P$ 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 $\bigsqcap\mathsf{H}$ and we let $\mathsf{h} \sqcap \mathsf{h}'$ equal $\sqcap\{\mathsf{h}, \mathsf{h}'\}$ for any two histories h and h'. The least upper bound (*lub*) is defined in the analogous manner. We write *lub* of H as $\bigsqcup\mathsf{H}$ and we let $\mathsf{h} \sqcup \mathsf{h}'$ equal $\sqcup\{\mathsf{h}, \mathsf{h}'\}$. 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 $\mathsf{h} \sqsubseteq \mathsf{g}$ and $\mathsf{h}' \sqsubseteq \mathsf{g}$. 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 $\mathsf{h} = \bot \bullet \sigma$

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 $\mathsf{v}_0 = \perp$, 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 ($\mathsf{k} \geqslant 1$), and let v_k be the value written by the k^{th} write. Further, let $\mathsf{v}_0 = \perp$.

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 $\mathsf{v}_\mathsf{k} = \mathsf{v}_0$ and (b) there is no $\mathsf{l} > \mathsf{k}$ 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 $\mathsf{v}_\mathsf{k} = \mathsf{x}$, (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* $\mathsf{n}-\mathsf{t}$ *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 $2\mathsf{k}$ 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 $\mathsf{n}-\mathsf{t}$ correct objects. If in a run r, a round rnd terminates based on replies from a set C of $\mathsf{n}-\mathsf{t}$ 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 $\mathsf{f} < \mathsf{n}/3$, 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 $\mathsf{f} < \mathsf{n}/3$, 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 $3\mathsf{P}$ 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 $3\mathsf{P}$. While Ω eventually outputs the same correct process to every correct process, $3\mathsf{P}$ 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 $3\,P$ suspects exactly the processes that have crashed initially.

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

Theorem 1 (Lower Bound)**.** *Assuming* $\mathsf{n} \leqslant 4\mathsf{f}$*, every one-step consensus algorithm* 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 $\mathsf{n} = 4$ 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 $\mathsf{k}-1$ steps. Since in each round a process waits for a message from at most $\mathsf{n}-\mathsf{f}$ processes, one entry is empty. The decision value is bracketed $(0)/(1)$.

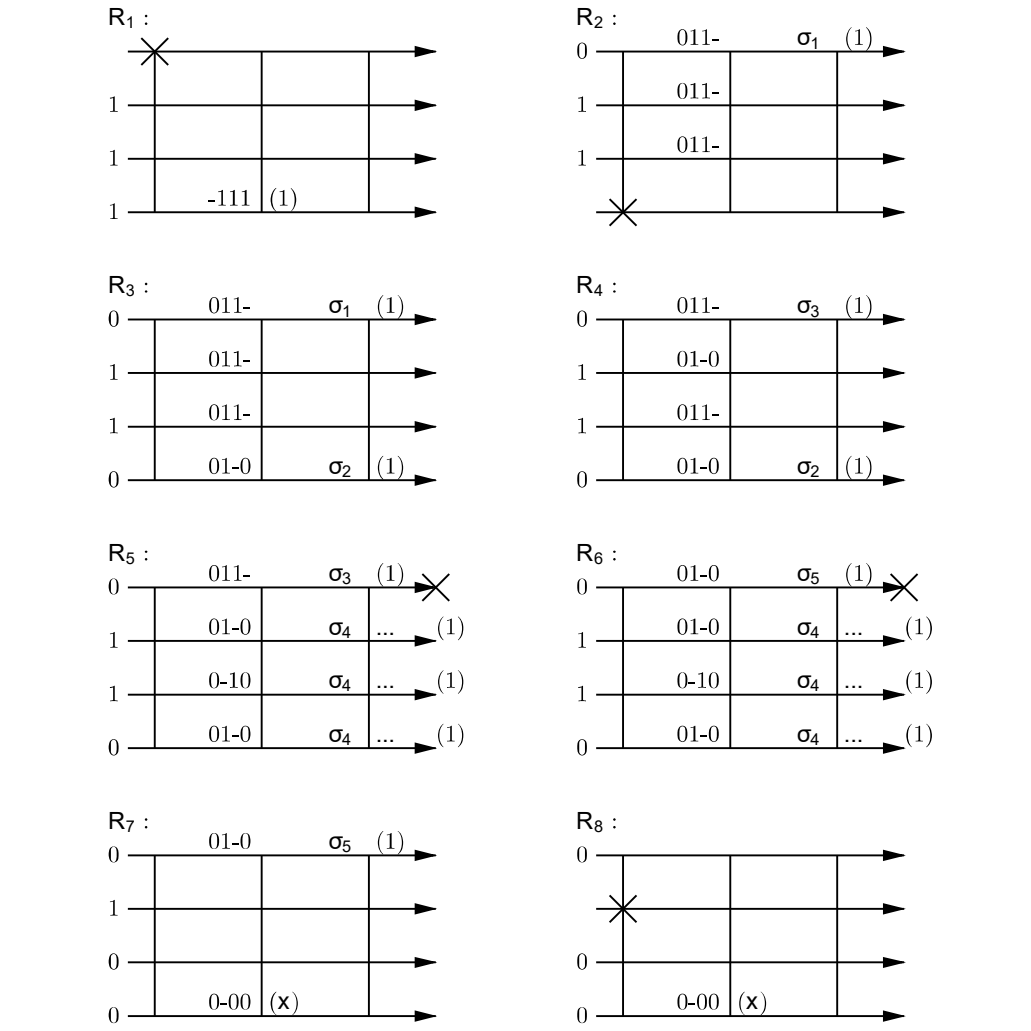
Two runs R_1 and R_2 are *similar* for process p up to step k, iff the state of p after k steps in R_1 is identical to the state of p after k steps in R_2. If two runs are similar for some process p, then p decides the same value in both runs.

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 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 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 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