# Fault-Tolerance in Storage-Centric Systems

Dissertation submitted for the degree of
"Doctor of Philosophy"
by

**Gregory V. Chockler**

Submitted to the Senate of the Hebrew University in Jerusalem (2003)

August 11, 2004

This work was carried out under the supervision of
**Prof. Dahlia Malkhi**.

# Contents

[1]This chapter is based on the PODC'02 paper by Chockler and Malkhi [CM02].

[2]This chapter is based on the paper by Chockler and Malkhi [CM03].
[3]This chapter is based on the paper by Chockler, Keidar and Malkhi [CKM03].

[4]This chapter is based on the WRSM'03 paper by Chockler, Malkhi, Merimovich and Rabinowitz [CMMR03].

# Abstract

This dissertation investigates the issues of reliability and high-availability in *storage-centric* distributed systems where a dynamic, possibly unbounded group of transient client processes utilize commonly accessible storage for achieving common goals (such as agreement and coordination). The storage-centric paradigm is especially suitable for designing Internet services featuring wide spread and high-degree of decentralization. It also faithfully models today's state-of-the-art *storage area network (SAN)* technology where storage units are directly attached to a high speed network that is accessible to clients.

Storage-centric systems are fundamentally different from traditional cluster-based environments. There are several reasons to this of which the most important are: (1) an inherent inability of the storage units to communicate, (2) a high dynamism of client processes, and (3) a possibility of storage unit failures in addition to the process failures. This calls for a completely new collection of methods and tools for achieving reliability in these systems. The main objectives of this dissertation are: (1) to develop a comprehensive methodology for supporting universal service replication in failure-prone storage-centric systems; (2) to investigate the inherent cost of this methodology under various failure models; and (3) to show its feasibility by implementing a practical fault-tolerant system based on the storage-centric design paradigm.

Our study of storage-centric environments is based on a precise mathematical formalism given by an asynchronous shared memory model with objects prone to *non-responsive* (NR) failures. According to this formalism, the shared storage units are modeled as shared objects, and clients are modeled as processes. We consider two types of shared object failures: a non-responsive crash (NR-crash) and non-responsive arbitrary (NR-arbitrary). The first type models benign faulty storage servers, whereas the second type models arbitrary, possibly malicious, storage failures. In these two failure models, we are interested in high-level object constructions that are (1) *fault-tolerant* in the sense that they *mask* the underlying object failures, and (2) *wait-free* in the sense that the high-level operations never get stuck despite any number of process (client) failures.

In this environment, we show a new agreement protocol that extends the well-known Paxos protocol of Lamport to support unbounded number of dynamic client processes. The protocol is resilient to any number of client failures and to a threshold of NR-crash or NR-arbitrary failures. It makes use of a shared memory object, called a *ranked register*, that promotes understanding and analysis of Paxos and of general coordination in distributed systems. To support coordination among unbounded number of clients, our ranked register object implementation relies on shared object with read-modify-write operations. The use of strong memory objects is necessitated by the existing space lower bounds of consensus implementations. It is further justified by the space complexity lower bound of ranked register object implementations presented in this thesis.

To ensure liveness, we complement our agreement protocol with an eventually-safe leader election primitive. For the first time, we provide an implementation of eventually-safe leader election from the bare shared memory environment, under an eventual partial synchrony assumption. Our implementation is simple, efficient, and provides for coordination among unbounded number of clients while utilizing a single shared register. It introduces as a building block the abstraction of an *eventual lease* that augments the time-based mutual exclusion algorithm of Fischer with expiration and renewals.

We investigate the cost of achieving optimal resilience ($t < n/3$) by fault-tolerant wait-free object emulations in the presence of NR-arbitrary storage failures. We show for the first time a wait-free safe register construction utilizing $n > 3t$ shared memory objects up to $t$ of which can incur NR-arbitrary failures. We also establish a fundamental tradeoff between the failure resilience and the number of communication rounds required to construct a safe register out of NR-arbitrary faulty objects.

Finally, we demonstrate the practical value of our replication methodology, by utilizing it in the implementation of a CORBA fault-tolerance [Obj00] infrastructure, called *Aquarius*. We evaluate the Aquarius' performance, and demonstrate its utility by implementing a fault-tolerant SQL database on top of it.

# Chapter 1

# Introduction

Only a decade ago, issues of replication, high availability and load balancing were the focus of small, closely coupled cluster projects. Consequently, techniques for cluster management and small replication systems are abundant. However, the advent of the Internet led to wide spread and highly decentralized access of services and content that calls for new paradigms and methods for reliable information storage and retrieval. One of such paradigms is a *storage-centric* distributed computing.

A storage-centric system consists of a fixed collection of *storage servers* a threshold of which may be faulty. Each server is responsible solely for implementing certain objects, e.g., a single shared register, that is accessible by any number of clients. This paradigm provides for coordination and information sharing among transient clients, possibly numerous, through the group of servers. It does not require servers to interact among themselves, and it avoids the complexity of failure monitoring and reconfiguration which is manifested, e.g., in group communication middlewares [Pow96, CKV01]. Several distributed systems existing today were designed according to the storage-centric paradigm. Examples include Fleet [MR00], SBQ-L [MAD02], Agile Store [LAV01], Coca [ZSR02], and [Baz00].

Another application domain where the storage-centric paradigm applies, reflects recent advances in hardware technology that have made possible a new approach for storage sharing, in which clients access disks directly over a *storage area network* (SAN). In a SAN, disks are directly attached to a high speed network that is accessible to clients. The clients access raw disk data, which is mediated by disk controllers with limited memory and CPU capabilities. Clients run file system services and name servers on top of raw I/O. Since clients (or a group of designated SAN servers) need to coordinate and secure their accesses to disks, they need to implement distributed access control and locking for the disks. However, once a client obtains access to a file, it accesses data directly through the SAN, thus eliminating the slowdown bottleneck at the file system server. IBM's *Storage Tank* [Bur00] is an example of a commercially available SAN system that solves many of the coordination, sharing and security issues involved with SANs. Other examples include Compaq's Petal [LT96] and Frangipani [TML97], Disk Paxos [GL03], and Active Disk Paxos [CM02]. The recently developed iSCSI protocol and standard [SMS$^+$03] enhance the storage device connectivity beyond the SAN boundaries leading to a new generation of storage-centric services with a truly global device sharing.

The storage-centric paradigm also represents a convenient framework for adding replication to existing Internet based client/server services for improved robustness and availability. The only necessary additions are a thin server-side wrap providing a facility for storing and retrieving temporary 'meta-data' and a client-side agent for coordinating the client access to the service replicas. In Chapter 8, we demonstrate utility of this method by employing it for seamless integration of fault-tolerance infrastructure into an existing CORBA [Obj99] based middleware.

This dissertation focuses on the issues of fault-tolerance and high availability in storage-centric systems. The thesis pursued by this dissertation is (1) to develop a comprehensive methodology for realizing universal service replication in failure-prone storage-centric systems, (2) to investigate the inherent cost of this methodology under various failure models, and (3) to show its feasibility by implementing a practical fault-tolerant system based on the storage-centric design paradigm.

## 1.1 Overview of the main results

Our study of storage-centric environments is based on a precise mathematical formalism given by an asynchronous shared memory model with objects prone to *non-responsive* (NR) failures. This formalism was first introduced by Jayanty et al. in [JCT98] (see Chapter 3 for a detailed description). According to this formalism, the shared storage units (servers) are modeled as shared objects, and clients are modeled as processes. From here on, we will use the terms storage units and shared objects interchangeably.

We consider two types of shared object failures: non-responsive crash (NR-crash) and non-responsive arbitrary (NR-arbitrary) failures. The first type models benign faulty storage servers, whereas the second type models arbitrary, possibly malicious, storage failures. In these two failure models, we will be interested in high-level object constructions that are (1) *fault-tolerant* in the sense that they *mask* the underlying object failures, and (2) *wait-free* in the sense that the high-level operations never get stuck despite of any number of process (client) failures.

For the NR-crash model, all our constructions achieve the optimal resilience of $t < n/2$ faulty objects. In the NR-arbitrary model, we also provide constructions that do not match the optimal resilience of $t < n/3$ faulty objects. This is motivated by the lower bounds shown in Chapter 6 that establishes an inherent tradeoff between the resilience and the number of communication rounds.

In addition, we will be interested in *uniform* constructions that do not depend on a priori knowledge of the number of clients and/or their identities. This requirement is important as it results in scalable constructions where the memory requirements of the underlying objects do not grow with the number of accessing clients.

In the following, we discuss in more detail the main results and contributions of this work.

### 1.1.1 Universal service replication

We provide a comprehensive framework for implementing universal service replication in storage-centric systems with fail-prone storage accessed by possibly unbounded universe of fail-prone clients.

At the core of the universal service replication is an *agreement protocol* that ensures that the operations which access (and possibly modify) the service state are applied in the same order at all replicas thus keeping their states consistent. However, in order to allow the agreement protocol to make progress, it is well known that the environment must be eventually synchronous for sufficiently long [FLP85]. Intuitively, this requisite enables a unique leader to be established and enforce a decision. This separation between safety and liveness is at the core of the Paxos approach [Lam98, Lam01a, PLL00, Lam96] to implementing universal service replication: First, provide an implementation of an agreement protocol that never violates safety but is prone to livelock when the system is unstable; Second, complement the always-safe agreement protocol with an eventually-safe leader election primitive. Our implementation of the universal service replication in storage-centric systems follows this decomposition. Below we discuss our implementations of agreement and leader election in more detail:

#### The always-safe agreement implementation

The always-safe but possibly-not-live agreement protocol of Paxos, called Synod, is a 3-phase commit protocol [Ske81, Ske82] that uses the 1st phase to determine a proposition value, the 2nd phase to fix a decision value, and the 3rd phase to commit to it. The Synod protocol was recently adapted for utilization in storage-centric (shared-memory) environments by the Disk Paxos (Disk Synod) protocol [GL03]. In this work we provide several important contributions that enhance this line of research:

- We provide an adaptation of Paxos that supports infinitely many clients;

- The memory complexity of our solution is constant;
- Our construction makes use of a shared memory object, called a *ranked register*, that promotes understanding and analysis of Paxos and of general coordination in distributed systems.
- Our agreement protocol is built directly over the ranked register object, providing for an efficient one-tier implementation;
- We prove a lower bound showing that read-modify-write objects are necessary for supporting infinitely many clients with bounded memory.

Our agreement protocol for the NR-crash model are presented in Chapter 4. In Chapter 7, we present an agreement protocol that tolerates NR-arbitrary object failures. The NR-crash resilient agreement protocol matches the optimal $t < n/2$ resilience bound. Although our NR-arbitrary resilient agreement protocol achieves only a sub-optimal resilience of $t < n/5$, it is efficient in terms of both the number of communication rounds and the server memory requirements. This is justified by the results of Chapter 6 that establish a fundamental tradeoff between the resilience and the number of communication rounds.

### The eventually-safe leader election implementation

Algorithms for mutual exclusion in the presence of failures must be based on timeliness assumptions, as they have to be able to attain progress in spite of process failures while executing in their critical section. However, while the shared memory literature is abundant with time-based mutual exclusion algorithms, to the best of our knowledge, the eventually-safe leader election was never considered in the shared memory context. In contrast, the problem of implementing leader election/failure detectors in message-passing systems has recently become an active research area (see e.g., [ADGFT03]). In Chapter 5 and in [CM03], we address for the first time, the problem of implementing eventually-safe leader election from the bare shared memory environment, under an eventual partial synchrony assumption. Our implementation is simple, efficient, and provides for coordination among a priori unknown number of clients while utilizing a single shared register.

Our approach to implementing an eventually-safe leader election introduces as a building block the abstraction of an *eventual lease*. Informally, a lease is a shared object that supports a CONTEND operation, such that when CONTEND returns 'true' at any process, it does not return 'true' to any other process for a pre-designated period. The lease automatically expires after the designated time period. In addition, our lease supports a RENEW operation which allows a non-faulty leader to remain in leadership (indefinitely).

A lease is substantially different from a mutual-exclusion object: By its very nature, it becomes possible for other processes to recover an acquired lease regardless of the actions of the others, including the case that the process that held the lease has failed.

Additionally, a lease may not be safe for an arbitrarily long period. Indeed, in an eventually (or intermittently) partially synchronous settings, any lease's pre-designated exclusion period entails no safety guarantee during periods of asynchrony. However, when the system stabilizes, all previous (possibly simultaneous) leases expire, and safety is recovered by the very nature of leases. Thus, despite any transient periods of instability, leases guarantee that once a system becomes synchronous for sufficiently long, it will be possible for processes to acquire exclusive leases. Renewals also provide automatic recovery: Only one renewal emerges successfully after system stabilization, despite any unstable past periods, and despite the possible existence of multiple simultaneous lease holders before the stability.

Our lease implementation utilizes a single wait-free shared multi-reader/multi-writer regular register [Lam86]. Such a register can be efficiently constructed out of $n > 2t$ base objects up to $t$ of which can incur NR-crash failures, and is possible (at higher cost) in the NR-arbitrary model.

### 1.1.2 The communication vs. resilience tradeoff for the NR-arbitrary model

Our construction of universal service replication in the NR-arbitrary model is efficient in terms of communication complexity, but has a low resilience threshold of $t < n/5$. The same low resilience vs. good communication complexity pattern can also be found in previous work that constructed wait-free objects in the NR-arbitrary model. For example, the safe register constructions of [MR00] and [JCT98] used $4t+1$ and $5t+1$ base objects respectively, and the universal service construction of [CMR01] used $6t+1$ base objects.

A natural question to ask is whether the resilience threshold $t < n/4$ (where $n$ is the number of fault-prone objects) is tight in this model. Intuitively, one would hope for a resilience bound of $t < n/3$. (It is easy to see that in this model it is impossible to obtain better resilience than $t < n/3$ [MAD02]). Several previous works have addressed this question, and have achieved better resilience by weakening the model in different ways – by adding synchrony [Baz00]; by storing authenticated (signed) self-verifying data at the servers [MR00, MAD02]; or by assuming that clients never fail and providing solutions that may block indefinitely if clients do fail [MAD02, ABO03]. However, $t < n/4$ is the best resilience threshold previously achieved for wait-free constructions in the NR-arbitrary model.

In Section 6.4.1 of Chapter 6, we prove that if $t \geq n/4$, then it is impossible to emulate the WRITE operations of a wait-free register by invoking a single round of operations on the base objects. Our proof applies to binary single-writer single-reader *safe registers*; the weakest meaningful register type [Lam86].

Moreover, in Section 6.4.2 of Chapter 6, we show that if $n = 3t + 1$, then any algorithm in which the reader does not modify the base objects' states may need to invoke as many as $t + 1$ rounds of read operations on base objects in order to emulate a single READ operation of a single-writer single-reader safe register. More generally, for any $0 \leq f \leq t$, there is a run in which $f$ objects are Byzantine faulty in which the algorithm invokes $min(t+1, f+2)$ rounds of base object operations.

The above results demonstrate that although the optimal failure threshold of $t < n/3$ is achievable, it incurs a significant overhead in terms of communication rounds invoked by the *read* and *write* implementations. Our universal service constructions as well as several the constructions of [JCT98, MR00, CMR01] avoid this overhead by lowering the resilience threshold.

### 1.1.3 Fault-tolerant CORBA using a storage-centric paradigm

To demonstrate the practical value of our replication methodology, we utilize it for implementing a CORBA fault-tolerance [Obj00] infrastructure, called *Aquarius*. The Aquarius architecture follows the storage-centric design paradigm: each object replica is managed by a thin wrap that provides a facility for storing and retrieving temporary 'meta-data' per object. Clients access replicas through stateless proxy servers. In order to coordinate updates to different copies of an object, proxies run a storage-centric agreement protocol utilizing object wraps as storage units.

The Aquarius architecture puts minimal additional functionality on object replicas, who neither communicate with one another, nor are aware of each other. Proxies are also not heavy. Their interaction is through rounds of remote invocations on quorums of object replicas. The design has several important advantages. First, it alleviates the cost of monitoring replicas and reconfiguration upon failures. Second, it provides complete flexibility and autonomy in choosing for each replicated object its replication group, failure threshold, quorum system, and so on. In contrast, most implementations of state machine replication pose a central total-ordering service which is responsible for all replication management (see, e.g., [Pow96, CKV01] for good surveys). Third, it allows support for Byzantine fault tolerance to be easily incorporated into the system by employing an NR-arbitrary agreement protocol combined with Masking Quorum systems [MR98] and response voting. Finally, limiting redundancy only to the places where it is really needed (namely, object replication) results in an infrastructure with only a few necessary components thus simplifying the system deployment and reducing its code complexity. (Our CORBA implementation uses 4K lines of Java code for each of the client and server implementations).

The design and implementation of Aquarius is described in detail in Chapter 8.

# Chapter 2

# Related Work

This dissertation describes the results of our investigation into the possibility and cost of building fault-tolerant services in storage-centric systems. The environment model that faithfully reflects the storage-centric setting is an asynchronous shared memory system where processes interact by means of a finite collection of shared objects some of which can be faulty. Computing in shared memory systems with faulty objects was studied in the works by Afek et al. [AGMT95] and Jayanti et al. [JCT98]. In particular, shared memory systems with non-responsive object failures considered in this dissertation were introduced in [JCT98].

Our study focuses on solving the Consensus problem [LSP82]. We discuss the Consensus problem and approaches to its solutions in Section 2.1.

Part I of this dissertation deals with solving Consensus in storage-centric environments with crash failure-prone storage with an emphasis on providing solutions which are oblivious to the number of participating client processes. More specifically, Chapter 4 focuses on guaranteeing safety of the Consensus decision in the presence of infinite number of processes. The existing work relevant to the results in Chapter 4 is discussed in Section 2.2.

The liveness of the Consensus algorithm in Chapter 4 depends on the availability of a failure detector of class $\Omega$ [CHT96] that guarantees that eventually all the correct processes permanently trust the same correct process as a leader. Chapter 5 deals with implementing an $\Omega$ failure detector in a partially synchronous shared memory model with a priori unknown number of participating processes. Our implementation borrows some of its ideas from the vast body of work on time-based mutual exclusion. We survey the prior work on this subject in Section 2.3.

In Part II of this thesis we turn to the treatment of Byzantine fault-prone storage. The results in Chapter 6 establish tight lower bounds on the round complexity of implementing a wait-free safe [Lam86] register from $n \geq 3t+1$ base objects upto $t$ of which can suffer from Byzantine failures. The existing work on emulating fault-tolerant objects from Byzantine fault-prone storage is discussed in Section 2.4.

In Chapter 7 we tackle the problem of implementing Consensus with Byzantine fault-prone storage. Our solution follows the deconstruction approach of Chapter 4 and achieves the $t < n/5$ failure resilience. Section 2.5 discusses the relevant existing work in this area.

Finally, Part III of this thesis shows an implementation of a storage-centric fault-tolerance infrastructure for distributed CORBA objects. CORBA fault-tolerance has received significant attention in recent years, both in research and standardization, culminating with the recently adopted Fault-Tolerant CORBA (FT-CORBA) standard [Obj00]. In Section 2.6 we survey the existing work on this subject.

## 2.1 The Consensus Problem

The Consensus problem [LSP82] is one of the most fundamental problems in distributed computing. Consensus is the building block for replication paradigms such as state machine replication [Lam78, Sch90], group membership (see [Pow96, CKV01] for survey), virtual synchrony [BJ87], atomic broadcast [CT96], total ordering of messages [KD00, FLS01], etc. Consensus is known to be unsolvable in most realistic models such as asynchronous message passing systems [FLP85] and asynchronous shared memory with read-write registers [LAA87, Her91, DDS87] if even a single process can fail by crashing. Wait-free consensus is also unsolvable in shared memory environments with non-responsive object failures [JCT98] regardless of the number, size and type of the shared objects used by the implementation. Consequently, it is unsolvable in storage-centric systems prone to non-responsive storage failures regardless of the functionality supported by storage units.

While it is usually straightforward to guarantee the consistency of a consensus decision alone (safety), the difficulty is in guaranteeing progress in face of uncertainty regarding process and storage failures. The usual approaches to circumventing Consensus impossibility include strengthening the basic model by assuming different degrees of synchrony (see e.g., [DDS87, DLS88, CF99]), augmenting the system with unreliable failure detectors [CT96], and employing randomization (see a survey in [CD89, Asp03]).

Our approach to solving Consensus uses one of the most widely deployed implementations of the state machine replication [Lam78, Sch90], the Paxos algorithm [Lam98, Lam01a, PLL00, Lam96]. At the core of Paxos is a consensus algorithm called *Synod*. The Synod protocol deals with the Consensus impossibility by guaranteeing progress only when the system is stable so that an accurate leader election is possible. This assumption is equivalent to assuming a failure detector of class $\Omega$ [CHT96] which was shown in [CHT96] to be the weakest failure detector that can be used to solve Consensus.

## 2.2 Consensus with Benign Faulty Storage and Infinitely Many Clients

Chapter 4 of this dissertation focuses on guaranteeing safety of the Consensus decision in the presence of infinite number of processes. Other results in this model and a classification based on levels of simultaneity can be found in [MT00, GMT01]. The work presented in this chapter was inspired by the Disk Paxos protocol of Gafni and Lamport [GL03] that provides an adaptation of the original Paxos algorithm to Storage Area Network (SAN) environments (the aspects of the SAN technology that are relevant to our study are surveyed in Section 2.2.1 below).

In Disk Paxos, the protocol state is replicated at network attached disks some of which can crash or become inaccessible. The participating processes access the state replicas directly over a SAN. Disk Paxos assumes simple commodity disks which support only primitive read and write operations. It supports a bounded and known number of clients, and uses disk memory proportional to their number. In contrast, we stipulate storage that is capable of serving higher semantics objects (such as storage servers or Active Disks [RFGN01, AUS98]), which provide us with the strength needed to guarantee safe decisions in face of an unbounded number of clients. The amount of memory we utilize per storage component is fixed regardless of the number of participating clients.

The Consensus solution in Chapter 4 first breaks the Paxos protocol using an abstraction of a shared object called a *ranked register*. The idea of decomposing the Paxos protocol in this way is due to Boichat et al. [BDFG01]. In particular, [BDFG01] proposes a modular deconstruction of Paxos based on a simple shared memory register called *round-based register*. Intuitively, both the round-based register and the ranked register objects encapsulate the notion of Paxos *ballots*[1] which are used by the protocol to ensure value consistency in presence of concurrent updates. While being in line with the general deconstruction idea of Boichat et al., our ranked register object nevertheless provides weaker guarantees and supports a slightly different interface. A detailed comparison is provided in Section 4.3. A different deconstruction of the Paxos protocol is provided in [BDFG03]. This deconstruction employs a more abstract shared-object definition, called $\Diamond Register$. The $\Diamond Register$ avoids referring to ranks (or rounds), and thus is a higher level abstraction that does not directly include implementation details. On the other hand, as discussed in [BDFG03], the $\Diamond Register$ admits inefficient implementations. This is prevented in the specification of our ranked register object.

[1]Ballots roughly correspond to *rounds* and to *ranks* in the round-based and the ranked register objects respectively.

### 2.2.1 SAN Technology

Our work on Consensus in storage-centric systems was motivated in part by advances in storage technology and the SAN paradigm. A storage area network enables cost-effective bandwidth scaling by allowing the data to be transferred directly from network attached disks to clients so that the file server bottleneck is eliminated. The Network Attached Secure Disks (NASD) [GNA$^+$98] of CMU is perhaps the most comprehensive joint academy-industry project which laid the technological foundation of network attached storage systems. NASD introduced the notion of an *object storage device (OSD)* which is a network attached disk that exports variable length "objects" instead of fixed size blocks. This move was enabled by recent advances in the Application Specific Integrated Circuit (ASIC) technology that allows for integration sophisticated special-purpose functionality into the disk controllers. The NASD project also addresses other aspects of the network attached disk technology such as file system support [GNA$^+$97], security [GGT97] and network protocols [GNA$^+$98].

Active Disks [RFGN01, AUS98] is a logical extension of the OSD concept which allows arbitrary application code to be downloaded and executed on disks. One of the applications of the active disks technology is enhancing disk functionality with specialized methods, such as atomic read-modify-write, that can be used for optimization and concurrency control of higher-level file systems.

Issues concerned with data management in SAN based file systems, such as synchronization, fault tolerance and security, are investigated in [Bur00] in the context of the IBM Storage Tank project.

Other work which addresses scalability and performance issues of network storage systems (not necessarily concerned with network attached disks) include NSIC's Network-Attached Storage Device project [Con], the Netstation project [HMF98] and the Swarm Scalable Storage System [HMS99]. Petal [LT96] is a project to research highly scalable block-level storage systems. Frangipani [TML97] is a scalable distributed file system built using Petal. xFS: Serverless Network File Service [ADN$^+$96] attempts to provide low latency, high bandwidth access to file system data by distributing the functionality of the server (e.g. cache coherence, locating data, and servicing disk requests) among the clients.

Concurrency control was identified as one of the critical issues in the network attached storage technology because of inherent lack of a central point of coordination [AGG00]. The concurrency control in the Petal [LT96] virtual disk storage system and the Frangipani [TML97] file system is achieved using replicated lock servers which utilize Paxos for consistency. Consequently, Disk Paxos is a natural candidate for enabling lock management in network attached storage systems. In this chapter we show that by enhancing network attached disk functionality with two simple read-modify-write operations, which are realistic to support with the OSD and Active Disk technologies, it is possible both to adapt Disk Paxos to support an unbounded number of clients and to reduce its communication cost.

## 2.3 Mutual Exclusion in Time-Based Shared-Memory Environments

The two most commonly used timing assumption in the context of time-based mutual exclusion are the *known delay model* of [AT99, AT96b, AT96a] and the *unknown delay model* of [AAT97]. The known delay model was first formally defined in [AT96b]. The first mutual exclusion algorithm explicitly based on the known delay assumption was the famous Fischer algorithm, which was first mentioned by Lamport in [Lam87]. In [Lam87], another timing based algorithm is presented. This algorithm assumes a known upper bound on time a process may spend in the critical section.

Alur et al. consider in [AAT97] the unknown delay model: The time it takes for a process to make a step is bounded but unknown to the processes. The paper presents algorithms for mutual exclusion and Consensus in this model. A remarkable feature of these algorithms is their ability to preserve safety even in completely asynchronous runs. However, they are guaranteed to satisfy progress only if the system behaves synchronously throughout the entire run. The mutual exclusion algorithm of [LS92] combines the ideas of Fischer and Lamport's fast mutual exclusion algorithm [Lam87] to derive a timing based algorithm that guarantees progress when the system stabilizes while being safe at all times. However, the algorithm of [LS92] is not fault-tolerant.

The eventually known delay ($\Diamond$ND) timing model considered in Chapter 5 is an extension of a standard asynchronous shared memory model to include partial synchrony assumptions. The $\Diamond$ND model assumes that there exist a non-negative real number $\rho < 1$ and a positive integer $\delta$ such that eventually, the process hardware clock drift rate equals $\rho$, and the time it takes for a process to complete a shared memory access is bounded by $\delta$. Note that the $\Diamond$ND model differs from the two delay models above as it assigns different time bounds for a local process step and a shared memory access. This distinction is important in our case for in a storage-centric setting, each shared object is emulated by a collection of object replicas stored at remote servers, and therefore, every shared memory access bears the cost of a remote access. The $\Diamond$ND model is closely related to several semi-synchronous message-passing models defined in the past. In particular, it is similar to a model with partially synchronous processes and communication which is one of the partially synchronous models considered in [DLS88]. It also resembles the timed asynchronous model of Cristian and Fetzer defined in [CF99]. An interesting future direction would be to provide a formal treatment of the

$\Diamond$ND model-based algorithms within the existing and emerging real-time modeling frameworks such as [AL94], or the Timed I/O Automata (TIOA) model of [DKKV03, KLSV04].

As far as we know the ($\Diamond$ND) model was never considered in the shared memory context. Most of the existing time-based mutual exclusion algorithms are either not fault-tolerant [AT96b, AT96a], or resilient only to the timing failures [LS92, AAT97]. The fault-tolerant (wait-free) timing based algorithms of [AT99] are not suitable for the $\Diamond$ND model as they might violate safety and/or liveness even during synchronous periods if the delay constraints do not hold right from the beginning of the run.

Other properties that are of interest to us is the ability of timing based algorithms to support exclusion among arbitrarily many client processes and to work with weaker registers and/or a small number thereof. The latter is particularly important in failure prone environments as in these environments the registers must be first emulated out of possibly faulty components. In this respect the original solution by Fischer is superior to all the other algorithms as it is based on a single multi-writer multi-reader register. In fact, as we show in this chapter, the register is only required to support regular semantics (in the sense of [SPW03]), and hence may be emulated efficiently even in a message passing setting. This solutions was therefore chosen as a basis for our lease implementation. The algorithms of [LS92] and [LS92] are also oblivious to the number of participants and use two and three shared atomic registers respectively.

The goodness of timing based mutual exclusion algorithms are frequently assessed in terms of their performance in contention free runs. In particular, a good algorithm is expected to avoid delay statements when there are no contention. The performance of the timing based algorithms under various levels of contention is analyzed in [GM02]. The paper examines (both analytically and in simulations) the expected throughput of timed based mutual exclusion algorithms under various statistical assumptions on the arrival rate and the service time. The question of further optimizing our solutions for contention free runs is left for future research.

The eventually accurate leader election primitive in Chapter 5 is constructed on top of light-weight *lease* objects which are useful synchronization primitives in their own right. Gray and Cheriton were the first to employ leases in [GC89] for constructing fault-tolerant distributed systems. Lampson advocates in [Lam96, Lam01b] the use of leases to improve the Paxos algorithm. Boichat et al. [BDG02] introduce asynchronous leases as an optimization to the atomic broadcast algorithms based on the rotating coordinator paradigm. Chockler et al. [CMR01] show a randomized backoff based algorithm for implementing leases in a setting similar to the $\Diamond$ND model of this chapter. However, the algorithm of [CMR01] guarantees progress only probabilistically, and relies on shared objects that can measure the passage of time. Finally, Cristian and Fetzer [CF99] show an implementation of leases in timed asynchronous message passing systems.

## 2.4 Implementing Fault-Tolerant Objects from Byzantine Fault-Prone Storage

The results in Chapter 6 establish tight lower bounds on the round complexity of implementing a wait-free safe [Lam86] register from $n \geq 3t+1$ base objects upto $t$ of which can suffer from Byzantine failures. Martin et al. showed in [MAD02] that $n \geq 3t+1$ base objects are necessary to implement a 1-writer-1-reader safe register. However, $t < n/4$ was the best resilience threshold previously achieved for wait-free constructions in data-centric models with Byzantine fault-prone storage: The wait-free safe register constructions of Malkhi and Reiter [MR00], and that of Jaynti et al. [JCT98] are resilient to at most $n/4$ and $n/5$ Byzantine base object failures respectively. Other works achieved better resilience by weakening the model in different ways – by adding synchrony [Baz00]; by storing authenticated (signed) self-verifying data at the servers [MR00, MAD02]; or by assuming that clients never fail and providing solutions that may block indefinitely if clients do fail [MAD02, ABO03].

## 2.5 Consensus with Byzantine Fault-Prone Storage

In Chapter 7 we turn to implementing Consensus with Byzantine fault-prone storage. Our solution follows the deconstruction approach of Chapter 4: It employs a Paxos-like agreement protocol based on a weaker variant of ranked register object combined with an unreliable failure detector of class $\Omega$. The resulting algorithm tolerates upto $t < n/5$ Byzantine faulty storage components. This result improves the Consensus algorithm in [CMR01] that required $n \geq 6t+1$ storage servers. The Byzantine Paxos protocol by Castro and Liskov [CL02] (see also [Lam01b]) achieves the optimal failure resilience of $t < n/3$. However, their algorithm incurs the cost of three message delays to reach consensus in stable runs. In contrast, our algorithm when transformed to a message-passing model incurs only two message delays in stable runs thus matching the lower bound of [KR03]. The recent work [GV04] suggest that this resilience vs. round complexity tradeoff is inherent.

## 2.6 Fault-Tolerant CORBA

CORBA fault-tolerance has received significant attention in recent years, both in research and standardization, culminating with the recently adopted Fault-Tolerant CORBA (FT-CORBA) standard [Obj00]. The existing fault-tolerant CORBA implementations rely on group communication services, such as membership and totally ordered multicast, for supporting consistent object replication. The systems differ mostly at the level at which the group communication support is introduced. Felber classifies in [Fel98] existing systems based on this criterion and identifies three design mainstreams: *integration*, *interception* and *service*. Below we briefly discuss these approaches and give system examples.

With the integration approach, the ORB is augmented with proprietary group communication protocols. The augmented ORB provides the means for organizing objects into groups and supports object references that designate object groups instead of individual objects. Client requests made with object group references are passed to the underlying group communication layer which disseminates them to the group members. The most prominent representatives of this approach are Electra [LM97] and Orbix+Isis [ION94].

With the interception approach, no modification to the ORB itself is required. Instead, a transparent interceptor is over-imposed on the standard operating system interface (system calls). This interceptor catches every call made by the ORB to the operating system and redirects it (if necessary) to a group communication toolkit. Thus, every client operation invoked on a replicated object is transparently passed to a group communication layer which multicasts it to the object replicas. The interception approach was introduced and implemented by the Eternal system [MMSN98].

With the service approach, group communication is supported through a well-defined set of interfaces implemented by service objects or libraries. This implies that in order for the application to use the service it has to either be linked with the service library, or pass requests to replicated objects through service objects. The service approach was adopted by Object Group Service (OGS) [FGS98, Fel98].

Among the above approaches, the integration and interception approaches are remarkable for their high degree of object replication transparency: It is indistinguishable from the point of view of the application programmer whether a particular invocation is targeted to an object group or to a single object. However, both of these approaches rely on proprietary enhancements to the environment, and hence are platform dependent: with the integration approach, the application code uses proprietary ORB features and therefore, is not portable; whereas with the interception approach, the interceptor code is not portable as it relies on non standard operating system features.

The service approach is less transparent compared to the other two. However, it offers superior portability as it is built on top of an ORB and therefore, can be easily ported to any CORBA compliant system. Another strong feature of this approach is its modularity. It allows for a clean separation between the interface and the implementation and therefore matches object-oriented design principles and closely follows the CORBA philosophy.

Two more recent proposals, Interoperable Replication Logic (IRL) [BM03] and the CORBA fault-tolerance service (FTS) of [FH02], do not clearly fall in any one of the above categories. IRL [BM03] proposes to confine the replication logic support (such as total ordering, membership, etc.) within a separate tier for which stronger timing properties can be assumed or enforced. This decouples the replication support from dealing with asynchrony inherent to wide area network environment thus removing constraints on individual object replica placement. IRL provides for high degree of modularity and transparency by allowing the replication support to be introduced and evolved with only minimal changes to the existing clients and object implementations. Aquarius borrows the idea of the three tier architecture from IRL. However, in contrast to IRL, the middle tier of Aquarius consists of independent entities that are not aware of each other and do not run any kind of distributed protocol among themselves. The advantages of this architecture are further discussed in Section 8.5.

The core idea of the FTS [FH02] proposal is to utilize the standard CORBA's Portable Object Adaptor (POA) for extending ORB with new features such as fault-tolerance. In particular, FTS introduces a Group Object Adaptor (GOA) which is an extension of POA that provides necessary hooks to support interaction of the standard request processing mechanism and an external group communication system. The resulting architecture combines efficiency of the integration approach with portability and interoperability of the service approach. Aquarius utilizes the object adaptor approach for implementing the server side of the replication support (see Section 8.5).

# Chapter 3

# The Model

We consider an asynchronous shared memory system consisting of processes interacting with each other by means of a finite collection of shared objects, $O_1, \ldots, O_n$. Objects and processes are modeled as I/O automata [LT89]. Every shared memory object has an initial state and a *sequential specification* defining the object's behavior when it is accessed sequentially. A sequence of operations on a shared object is *legal* if it belongs to the sequential specification of that object. E.g., a sequence of operations on a read/write shared object is legal if each read operation returns the value written by the most recent write operation if such exists, or an initial value otherwise. Processes and objects are modeled as state machines. A *concurrent system* is a composition of a collection of process and object state machines. The *state* of a concurrent system is a vector of states, reflecting the states of the processes and objects in the system.

Operations on memory objects have non-zero duration, commencing with an invocation and ending with a response. The event $invoke(P_i, op, O_j)$ ($respond(P_i, res, O_j)$) models the invocation (resp. response) of an operation *op* on an object $O_j$ by a process $P_i$. A *run* of a concurrent system is a sequence (finite or infinite) of alternating states and events $s_0e_1s_1s_2e_2\ldots$ such that: (1) $s_0$ is an initial state; and (2) $s_i$, $i > 0$, is the result of applying $e_i$ to each component of $s_{i-1}$.

A *history* of a given run $\alpha$ is the subsequence of events in $\alpha$, the history of an object $O_j$ in $\alpha$ is the subsequence consisting of only the invocations and respond events of $O_j$, and the history of a process $P_i$ in $\alpha$ is the subsequence consisting of only the invocations and respond events of $P_i$. An invocation request $I = invoke(P_i, op, O_j)$ in $\alpha$ is said to be *complete* in $\alpha$, if $\alpha$ includes its matching $respond(P_i, res, O_j)$ event. Otherwise, $I$ is said to be *incomplete* in $\alpha$. An event $e$ is said to be *enabled* in a state $s$ if there is a transition from $s$ labeled with $e$. A run is *fair* if every event that is continuously enabled, eventually occurs. A run $\alpha$ is called *well-formed* if the history of every process $P_i$ in $\alpha$ consists of alternating invocations and matching responses, beginning with an invocation. A run $\alpha$ is called *weakly well-formed* if for every process $P_i$ and object $O_j$, the history of $P_i$ in $\alpha$ consists of alternating $invoke(P_i, op, O_j)$ and $respond(P_i, res, O_j)$ events, beginning with $invoke(P_i, op, O_j)$.

In the rest of this dissertation, we consider only well-formed, or when multiple object replicas need to be accessed in parallel, weakly well-formed runs (see Section 3.2 below). In addition, fair runs will be assumed in the proofs of liveness properties.

## 3.1 The failure modes

Processes may fail by stopping (crashing). The implementation should be *wait-free* in the sense that the progress of each non-faulty process should not be prevented by other processes concurrently accessing the memory nor by failures incurred by other processes. The shared memory objects may suffer two types of failures [JCT98]: *non-responsive crash (NR-crash)* and *non-responsive arbitrary (NR-arbitrary)* failures. An object that incurs a NR-crash failure behaves correctly until it fails, and once it fails it never responds to any invocation. An object that incurs a NR-arbitrary exhibits completely unrestricted behavior. Such an object may fail to respond to an invocation, or may respond with an arbitrary value.

## 3.2 Accessing objects in the presence of non-responsive failures

A process invoking an operation on an object remains blocked until the object responds. Thus, if a process is restricted to a single thread of control, when it invokes an operation on an object that fails to respond, the process remains blocked indefinitely. This makes it impossible to construct fault-tolerant implementations in the presence of non-responsive object failures. To circumvent this problem, we allow processes to consist of more than one thread of control. A common object access pattern (used throughout this dissertation) is to invoke an operation *op* concurrently on each object in a set $S$, and then wait for some of the invoked objects to respond. This can be done by having a main thread spawn off additional threads to handle the invocations of *op* on the different objects. In the rest of this dissertation, we omit the details of implementing concurrent object invocations using multiple threads, and simply say that a process invokes an operation on multiple objects. Although multi-threading allows a process to initiate several invocations concurrently on *different* objects, a process is still not allowed to initiate more than one invocation concurrently on the same object: i.e., each run is required to satisfy the weak well-formedness requirement above.

Note that due to asynchrony, a process cannot know the current status of invocations that were initiated in separate threads until they return. That is, when a process $P_i$ spawns a thread $t_j$ to invoke an operation *op* on an object $O_j$ the following three situations are indistinguishable to $P_i$: (1) the *invoke* event has not yet occurred (because $t_j$ is slow); (2) $invoke(P_i, op, O_j)$ has occurred, but $O_j$ did not yet respond (because $O_j$ is slow); and (3) $O_j$ has failed.

# Part I

# Tolerating Benign Failures

# Chapter 4

# Universal Service Replication with Benign Faulty Storage[1]

## 4.1 Introduction

In this chapter we present a solution for universal service replication using a storage-centric approach. In this approach, a highly available service is implemented by a replicated set of servers, a threshold of which may be faulty. Each server is responsible solely for implementing certain objects, e.g., a single shared register, that is accessible by any number of clients. Our paradigm provides for coordination and information sharing among transient clients, possibly numerous, through the group of servers. It does not require servers to interact among themselves, and it avoids the complexity of failure monitoring and reconfiguration which is manifested, e.g., in group communication middlewares [Pow96, CKV01].

From here on, we refer to shared storage units in our storage-centric system simply as *objects*. As in many other distributed settings, a fundamental enabler in this environment for clients to coordinate their actions is an agreement protocol. It is well known that in order to solve agreement in a non-blocking manner three phases are needed [Ske81, Ske82]. This leads to the usage of the Paxos protocol [Lam98, Lam01a, PLL00, Lam96] and its variants, as is done, e.g., in Petal [LT96] and Frangipani [TML97]. Briefly, the Paxos protocol is a 3-phase commit protocol that uses the 1st phase to determine a proposition value, the 2nd phase to fix a decision value, and the 3rd phase to commit to it. The Paxos protocol was recently adapted for utilization in the shared-memory model in the Disk Paxos protocol [GL03]. In this work we provide several important contributions that enhance this line of research:

- We provide an adaptation of Paxos that supports infinitely many clients;

[1]This chapter is based on the PODC'02 paper by Chockler and Malkhi [CM02].

- The memory complexity of our solution is constant;
- Our construction makes use of a new shared memory object, called a *ranked register*, that promotes understanding and analysis of Paxos and of general coordination in distributed systems.
- Our agreement protocol is built directly over the ranked register object, providing for an efficient one-tier implementation.

Of the above contributions, the most tangible one is the extension to support infinitely many clients. Both the original Paxos protocol and its Disk variant are geared toward a fixed and known number of clients. In particular, in Disk Paxos, each client must use a pre-designated memory to write values, and must read the values written by all other potential clients. Consequently, adding new clients to the system is a costly operation that involves real-time locking [GL03]. Also, the complexity of memory (disk) operations is linear in the number of clients.

In contrast, our solution is ignorant of the number of participating clients and their identities. In its core, it consists of a Paxos-like consensus algorithm built over a ranked register object, combined with an unreliable leader oracle which is a failure detector of class $\Omega$ [CHT96]. It utilizes read-modify-write shared objects to support unbounded number of clients. Our use of strong memory objects is justified by the result in Sections 4.6.1 and 4.6.2: In Section 4.6.1, we show that $\Omega(\sqrt{n})$ $n$-writer-$n$-reader registers are required to implement a wait-free $n$-process consensus in an asynchronous shared memory system augmented with a failure detector in $\Omega$; and the lower bound in Section 4.6.2 proves that at least $\Omega(n)$ shared $n$-writer-$n$-reader registers are required to implement the ranked register object in failure-free runs. Hence, to circumvent these lower bounds and provide a solution which is independent of the number of participating clients, we employ read-modify-write memory objects.

Note that the strengthened memory model is justified in practice. First, servers may support arbitrarily complex object semantics, and as for disks, this approach is motivated by recent development in controller logic that enhances the functionality of disks for SAN and provide for *Active Disks*, capable of supporting stronger semantics objects (see, e.g., [GNA$^{+}$98]). In particular, specialized functions that require specific semantics not normally provided by drives can be provided by remote functions on Active Disks. Examples include a *read-modify-write* operation, or an atomic *create* that both creates a new file object and updates the corresponding directory object. Such advanced operations are already used for optimization of higher-level file systems such as NFS on NASD [GNA$^{+}$97].

The existence of strong shared memory objects does not obviate the need for an agreement protocol. Admittedly, if we had even one reliable read-modify-write object, we could leverage coordination off it to solve agreement, as shown by Herlihy in [Her91]. However, objects stored by servers or disks could become unavailable. Unfortunately, it is impossible to use a collection of fail-prone read-modify-write objects to emulate a reliable one [JCT98]. Hence, our construction is necessarily more involved. It should be noted that using a collection of shared objects also has the benefit beyond high availability. Even in the case that disks are considered reliable, distributing client accesses among multiple objects prevents unnecessary contention. Hence, our solution provides for both high availability, and for load sharing among storage servers.

Our solution first breaks the Paxos protocol using an abstraction of a shared object called a *ranked register*, which is driven by a recent deconstruction of Paxos by Boichat et al. in [BDFG01]. (We compare our ranked register object with the round-based register of [BDFG01] in Section 4.3). Briefly, a ranked register object supports rr-*read* and rr-*write* operations that are both parameterized by a *rank* whose values are taken from a totally ordered set fixed in advance (e.g., the Paxos *ballots* are integers). The main property of this object is that a rr-*read* with rank $r_1$ is guaranteed to "see" any completed rr-*write* whose rank $r_2$ satisfies $r_1 > r_2$. In order for this property to be satisfied, some lower ranked rr-*write* operations that are invoked after a rr-*read* has returned must *abort*. Armed with this abstract shared object, we show the following two constructions:

1. We provide a simple implementation of Paxos-like agreement using the abstraction of one reliable shared ranked register object that supports infinitely many clients. Briefly, in these implementations a participating client chooses a (unique) rank, rr-*read* s the ranked register with it, and then writes the ranked register either with the value it read (if exists) or with its own input. If the rr-*write* operation succeeds (i.e., it does not abort), then the process decides on the written value. Else, it retries with a higher rank.

2. The reliable shared ranked register object cannot be supported for an unbounded number of clients using only finite read/write memory (proof is provided in Section 4.6.2). Furthermore, no single fail-prone object may implement it. Therefore, we provide an implementation of a ranked register object shared among an unbounded number of clients. The implementation employs a collection of read-modify-write registers, of which a threshold may become non-responsive. The fault tolerant emulation performs each rr-*read* or rr-*write* operation on a majority of the disks, and takes the maximally ranked result as the response from an operation. The number of objects required for the emulation is determined only by the level of desired fault tolerance, regardless of the number of participating clients.

Our approach is readily implementable in a SAN with Active disks. To this extent, it may serve as an important specification of the kind of functionality that is desired by SAN clients and that disk manufacturers may choose to provide. Additionally, our approach faithfully represents another realistic setting, the classic client-server model, with a potentially very large and dynamic set of clients. This is the setting for which scalable systems like the Fleet object repository [MR00] were designed. We advocate the storage-centric approach in more detail in two recent position papers [CMD03, Mal02].

## 4.2 System model

We consider an asynchronous shared memory system consisting of a countable collection of client processes interacting with each other by means of a finite collection of shared objects. The processes are designated by numbers $1, 2, \ldots$. Clients may fail by stopping (crashing). The implementation should be wait-free in the sense that the progress of each non-faulty client should not be prevented by other clients concurrently accessing the memory as well as by failures incurred by other clients. The failure model for the shared memory objects if NR-crash.

According to [JCT98], wait-free consensus is impossible in such a setting. This result holds regardless of the number, size and type of the shared objects used by the implementation. Therefore, similar to the Paxos approach, we overcome this impossibility by augmenting the system with a leader oracle. The oracle guarantees the eventual emergence of a unique non-faulty leader, though when this happens is unknown to the clients themselves.

We employ a distributed Boolean leader oracle $\mathcal{L}$ defined as follows: Each process $i$ accesses $\mathcal{L}$ via its local module $\mathcal{L}_i$ whose output at any given time is a Boolean value (true or false) indicating whether the process $i$ trusts itself as a unique leader. A Boolean leader oracle guarantees that the following property holds eventually:

**Property 1 (Unique Leader).** *There exists a correct process $i$ such that $\mathcal{L}_i$ permanently outputs true, and for each process $j \neq i$, $\mathcal{L}_j$ permanently outputs $false$.*

Note that Boolean leader oracles differ from more common $\Omega$[CHT96, LH94] failure detectors since the output of a failure detector in $\Omega$ is the trusted process identifier and not a Boolean value. Although, it is easy to see that the two classes are equivalent in a shared memory environment with read/write registers[2], Boolean leader oracles are nevertheless,

[2] $\mathcal{L}$ is derived from $D \in \Omega$ by having each process $i$ to output true if the $D_i$ outputs $i$, and false otherwise. In the opposite direction, $D \in \Omega$ can be simulated from $\mathcal{L}$ by having each process $i$ such that $\mathcal{L}_i$ outputs true, to write $i$ to a shared multi-writer/multi-reader read/write register $X$, and output the value read from $X$.

more suitable for our setting as they may have anonymous client processes whose number is unlimited and unknown.

## 4.3 The Ranked Register

Our Consensus object construction (see Section 4.4) employ a special type of shared memory object, called a *ranked register*, which for now we assume is failure-free. In Section 4.5, we show how to implement a fault tolerant ranked register object.

Intuitively, the ranked register object encapsulates the notion of *ballots* which are used by the Paxos protocol to ensure value consistency in presence of concurrent updates. The idea of modeling the Paxos protocol this way is due to [BDFG01]. However, while the ranked register object interface bears similarities to the *round-based register* of [BDFG01], its specification is weaker than that of [BDFG01]. We discuss the differences below. The register provides a clean isolation of the essential properties of Paxos into a well-defined building block, thus simplifying reasoning about the protocol behavior.

We now give a formal specification of the ranked register object. Let *Ranks* be a totally ordered set of ranks with a distinguished initial rank $r_0$ such that for each $r \in Ranks$, $r > r_0$; and *Vals* be a set of values with a distinguished initial value $v_0$. We also consider the set of pairs denoted *RVals* which is $Ranks \times Vals$ with selectors $rank$ and $value$. A ranked register is a multi-reader, multi-writer shared memory object with two operations: $\mathsf{rr}\text{-}read(r)_i$ by process $i$, $r \in Ranks$, whose corresponding response is $value(V)_i$, where $V \in RVals$. And $\mathsf{rr}\text{-}write(V)_i$ by process $i$, $V \in RVals$, whose reply is either $commit_i$ or $abort_i$. Note that in contrast to a standard read/write register interface, both rr-*read* and rr-*write* operations on a ranked register take a rank as an additional argument; and its rr-*write* operation might abort, whereas the *write* operation on a standard read/write register always commits (i.e., returns *ack*).

In the following discussion we often say that a rr-*read* operation $R$ *returns* a value $V$ meaning that the register responds with $value(V)$ in response to $R$. We also say that a rr-*write* operation $W$ *commits* (*aborts*) if the register responds with *commit* (*abort*) in response to $W$.

For simplicity, we assume that each run starts with $W_0 = \mathsf{rr}\text{-}write(\langle r_0, \perp\rangle)$ which commits. Furthermore, we will restrict our attention to runs in which invocations of rr-*write* on a ranked register use unique ranks. More formally, we will henceforth assume that all runs satisfy the following:

**Definition 1.** *We say that a run satisfies* rank uniqueness *if for every rank $r \in Ranks$, there exists at most one $v \in Vals$ and one process $i$ such that $\mathsf{rr}\text{-}write(\langle r, v\rangle)_i$ is invoked in the* *run.*

In practice, rank uniqueness can be easily ensured by choosing ranks based on unique process identifier and a sequence number. The main reason we use this restriction is to simplify establishing the correspondence between the values written with specific ranks and the values returned by the rr-*read* operation.

We now give a formal specification of the ranked register object. We start by introducing the following definition:

**Definition 2.** *We say that a rr-read operation $R = \text{rr-}read(r_2)_i$* sees *a rr-write operation $W = \text{rr-}write(\langle r_1, v\rangle)_j$ if $R$ returns $\langle r', v'\rangle$ where $r' \geq r_1$.*

The ranked register object is required to satisfy the following three properties:

**Property 2 (Safety).** *Every rr-read operation returns a value and rank that was written in some rr-write invocation. Additionally, let $W = \text{rr-}write(\langle r_1, v\rangle)_i$ be a rr-write operation that commits, and let $R = \text{rr-}read(r_2)_j$, such that $r_2 > r_1$. Then $R$ sees $W$.*

**Property 3 (Non-Triviality).** *If a rr-write operation $W$ invoked with the rank $r_1$ aborts, then there exists a rr-read (rr-write) operation with rank $r_2 > r_1$ which is invoked before $W$ returns.*

**Property 4 (Liveness).** *If an operation (rr-read or rr-write) is invoked by a non-faulty process, then it eventually returns.*

Allowing rr-*write* to abort sometimes is crucial for its implementability. Suppose that a rr-*read* operation with rank $r$ returns a value written by a rr-*write* operation with a rank $r' < r$. Later, when a subsequent rr-*write* with a rank $r' < r'' < r$ is invoked, it must abort due to this rr-*read*.

Also note that our ranked register object specification is very weak: In particular, it allows in some situations for rr-*write* operation to commit even though there exists another previously committed rr-*write* with a higher rank. The reason for that not being a problem stems from the way the ranked register object is used by the Consensus implementation in Section 4.4.1. In particular, each process in our Consensus implementation invokes rr-*write* only after it invokes rr-*read* with the same rank and this rr-*read* returns. Thus, the ranked register Safety property ensures that in every finite execution prefix, each value written by a committed rr-*write* must be returned by one of the rr-*read* operations with a higher rank if such exist. Consequently, in each run of the Consensus implementation, any rr-*write* operation, which is invoked after rr-*read* with a higher rank has returned, would necessarily abort.

Our specification of ranked register is weaker compared with the round-based register of [BDFG01]. The round-based register uses notions of partial operation ordering in the definition of the *write-commit* property ("if write$(k, v)$ commits, and no **subsequent** write$(k', v')$ with $k' \geq k$ and $v' \neq v$ is invoked, then any read$(k'')$ that commits, commits with $v$ if $k'' > k$", stressed text added here for clarity). To see that this definition is too strong, consider the following scenario. Suppose that a write $w_1 = write(k_1, v_1)$ is invoked and is still in progress when another write is invoked, $w_2 = write(k_2, v_2)$, with $k_1 > k_2$, $v_1 \neq v_2$. In this case, $w_2$ may commit. However, a subsequent read may "see" $w_1$, and the value of $w_1$ may be returned, contradictory to the requirement. In fact, the distributed implementation of round-based register in [BDFG01] does not prevent this. Moreover, it does not seem possible to prevent this in our setting. Finally, we should note that just by dropping 'subsequent' from the specification results in different problems. It is our view that there is no easy way to form the ranked register object specification using operation ordering, and hence, the specification above is qualitatively different from that of the round-based register.

## 4.4 The Consensus implementation

In this section we present the implementation of Consensus based on the ranked register object defined in the previous section. The algorithms in this section use the ranked register object as a black box.

In addition to a shared ranked register, our algorithms also employ atomic shared registers. It should be noted that these objects can be implemented in our models in a similar manner to the ranked register implementation, and hence we omit their explicit constructions.

### 4.4.1 Consensus using a ranked register object

We now outline an agreement protocol which employs a shared ranked register object. The pseudocode of the Consensus implementation is depicted in Figure 4.1. Each process $i$ iterates through the following steps until the decision is reached: First, $i$ checks whether some process has decided and written the agreement value into the *decision* register. If yes, this value is returned. Otherwise, $i$ calls a local procedure, chooseRank which is assumed to output monotonically increasing values $r \in Ranks$, and then waits until the output of $\mathcal{L}_i$ becomes *true*. Once this happens, the local DECIDE routine is invoked. It takes as arguments $i$'s initial value and the chosen rank. It returns the agreement value or aborts. The DECIDE routine is guaranteed to return an agreement value at the latest when a non-faulty leader has been elected and allowed to force a decision (i.e., Property 1 holds).

```
Shared: Ranked registers $rr$ initialized by rr-$write(\langle r_0, \perp\rangle)$ which commits;
Regular multi-writer/multi-reader register $decision$,
    with values in $RVals$, initialized by $write(\langle r_0, \perp\rangle)$
Local: $V \in RVals \cup \{abort\}$, $r \in Ranks$;

Process $i$:

propose($v$), $Vals \rightarrow Vals$
    $r \leftarrow r_0$;
    while($true$) do
        $V \leftarrow decision.read()$;
        if ($V.value \neq \perp$)
            return $V.value$;
        if ($\mathcal{L}_i$ = true) then
            $r \leftarrow$ chooseRank($r$);
            $V \leftarrow$ DECIDE($\langle r, v\rangle$);
            if ($V \neq abort$)
                return $V.value$;
        fi
    od

Function DECIDE($\langle r, v\rangle$), $RVals \rightarrow RVals \cup \{abort\}$:
    $V \leftarrow rr.$rr-$read(r)_i$;
    if ($V.value = \perp$) then
        $V.value \leftarrow v$;
    $V.rank \leftarrow r$;
    if ($rr.$rr-$write(V)_i = commit$) then
        $decision.write(V)$;
        return $V$;
    fi
    return $abort$;
```

Figure 4.1: Consensus using a ranked register object

We now outline the correctness argument of the agreement algorithm. Recall that $W_0$ is an initialization rr-*write* operation, assumed to commit at the start of any execution. Ignoring this initialization, the next lemma shows that once a consensus value commits, it remains fixed as the decision value throughout the execution.

**Lemma 1.** *For any finite execution $\alpha$, let $W_1 = rr.$rr-write$(\langle r_1, v_1\rangle)$, $W_1 \neq W_0$ be the lowest ranked* rr-*write invocation which commits in $\alpha$. Then, in any extension of $\alpha$ in which $W = rr.$rr-write$(\langle r, v\rangle)$, $r > r_1$, is invoked, $v = v_1$.*

*Proof.* Our proof strategy is to build a chain of rr-*write*'s from $W_1$ to $W$, such that each $W$ writes the value that it reads from the preceding rr-*write* in the chain. We then show that the same value is written in all of these rr-*write*'s by induction on the length of such chains.

Indeed, let $R = rr.$rr-*read*$(r)$ be the rr-*read* corresponding to $W$ that is executed before $W$ is invoked. By safety, $R$ returns the pair $\langle r_1, w_1\rangle$ or a higher ranking pair $\langle r_k, w_k\rangle$ that was written in some $W_k =$ rr-*write*$(r_k, w_k)$. Since $r_k > r_1$, again the corresponding $rr.$rr-*read*$(r_k)$ returns $\langle r_1, w_1\rangle$ or a higher ranked written value. And so on. Eventually, we obtain a unique chain $W_1, W_2, ..., W_k, W$, such that for each of $W_2, .., W_k, W$, the corresponding rr-*read* returns the value/rank pair written by the preceding rr-*write* in the chain.

We now show by induction on the length $k$ of the chain that $W$ writes $v_1$. If $k = 1$, then $R$ returns $v_1$ and by the agreement protocol $W$ writes $v_1$.

Otherwise, suppose for all chains of length $< k$ it holds that the last rr-*write* writes $v_1$, and consider the chain above of length $k$. For $W_k$, the (unique) chain from $W_1$ is $W_1, W_2, ..., W_k$. By the induction hypothesis, $W_k$ writes $v_1$. Hence, again $R$ reads $v_1$ and according to the protocol, $W$ writes $v_1$. □

The following theorem immediately follows from Lemma 1 (and the protocol):

**Theorem 1.** *The algorithm in Figure 4.1 guarantees that for any two processes $i$ and $j$ such that* propose$(v)_i$ *returns $V$ and* propose$(v')_j$ *returns $V'$ , $V = V'$; and $V$ is the argument of some* propose *operation which was invoked in the run.*

Next, we show liveness.

**Theorem 2.** *If some correct process invokes* propose*, then eventually all correct processes decide.*

*Proof.* First note that the atomic register semantics imply that once some process decides and completes its write operation to the *decision* register, all other process will eventually read this value and decide.

Otherwise, by definition of $\mathcal{L}$, there exists time $T$ such that Property 1 holds at all times $t > T$. Assume that *decision* is not written before $T$. Since by the theorem precondition, at least one correct process is taking steps after $T$, Property 1 implies that there exists a correct process $i$ such that at all times $t > T$, $\mathcal{L}_i$ outputs true, and for all $j \neq i$, $\mathcal{L}_j$ outputs false.

By non-triviality of the ranked register object, rr-*write* is guaranteed to commit once it is called with a rank which is the highest among all the ranks ever chosen by any process in the system. Since chooseRank returns monotonically increasing ranks, such a rank is eventually returned by chooseRank at $i$. Once, rr-*write* commits, $i$ writes the committed value to the *decision* register. Once this happens, all the correct processes eventually decide. □

## 4.5 Implementing a ranked register object

In this section, we deal with the problem of implementing a wait-free shared ranked register object. First, in Section 4.5.1, we specify how a single ranked register object is implemented from a read-modify-write object. Second, in Section 4.5.2, we present a wait-free self-construction of the ranked register object for the NR-Crash failure model.

### 4.5.1 A single ranked register object

Our shared memory model assumes the existence of atomic shared objects such as read-modify-write registers. By this, we capture the assumption that each "disk" is capable of accepting from clients subroutines with I/O operations for execution, and indivisibly performing them. The disk itself may become unavailable, and hence, the shared memory objects it provides may suffer non-responsive crash faults. For this reason, no single read-modify-write object suffices for solving agreement on its own (as in Herlihy's consensus hierarchy, see [Her91]). Rather, we first use each read-modify-write object to construct a ranked register object (which may also incur a non-responsive crash fault), and then, use a collection of ranked register objects to construct a non-faulty ranked register object, from which agreement is built.

Let $X = (\mathit{Ranks} \times \mathit{Ranks} \times Vals) \cup \{\langle r_0, r_0, \perp\rangle\}$ with selectors $rR$, $wR$ and $val$. The implementation of a ranked register object uses a single read-modify-write shared object $x \in X$ of unbounded size whose field $x.rR$ holds the maximum rank with which a rr-*read* operation has been invoked; $x.wR$ holds the maximum rank with which a rr-*write* operation has been invoked; and $x.val$ holds the current register value. The implementation pseudocode is depicted in Figure 4.2. It is quite straight-forward: read returns the current value of the register, and records its own rank. Write checks whether a higher ranking read was invoked, aborts if yes, and if not, modifies the value of the register and records its own rank. For clarity, invocations of read-modify-write operations rmw-*read*and rmw-*write*are enclosed within "lock" and "unlock" statements, to indicate that they execute indivisibly.

**Lemma 2.** *The pseudocode in Figure 4.2 satisfies Safety.*

Types: $X = (Ranks \times Ranks \times Vals) \cup \{\langle r_0, r_0, \perp\rangle\}$ with selectors $rR$, $wR$ and $val$
Shared: $x \in X$.
Initially $x = \langle r_0, r_0, \perp\rangle$

Local: $V \in RVals$, $status \in \{ack, nack\}$.

Process $i$:

```
rr-$read(r)_i$:
    lock $x$:
        $V \leftarrow$ rmw-$read(r)$
    unlock $x$
    return $V$

rr-$write(\langle r, v\rangle)_i$:
    lock $x$:
        $status \leftarrow$ rmw-$write(r, v)$
    unlock $x$
    if $(status = ack)$
        return $commit$
    return $abort$
```

```
Read-modify-write procedures:
rmw-$read(r)$:
    if $(x.rR < r)$
        $x.rR \leftarrow r$
    return $\langle x.wR, x.val\rangle$

rmw-$write(r, v)$:
    if $(x.rR \le r \wedge x.wR < r)$
        $x.wR \leftarrow r$
        $x.val \leftarrow v$
        return $ack$
    return $nack$
```

Figure 4.2: An implementation of a single ranked register object

*Proof.* That a rr-*read* operation can only return a valid value that was actually used in a rr-*write* operation or $\langle r_0, \perp\rangle$ is obvious from the code. Now consider a rr-*write* operation $W_1 =$ rr-$write(\langle r_1, v_1\rangle)_i$ that commits and let $R_2 =$ rr-$read(r_2)_j$, $r_2 > r_1$ be a rr-*read* operation which returns $\langle r, v\rangle$. Let $mw_1$ denote the rmw-$write()$ procedure called from within $W_1$ and $mr_2$ the rmw-$read()$ procedure invoked within $R_2$. Since the read-modify-write semantics of $x$ ensures sequential access, $mr_2$ must be sequenced after $mw_1$. For otherwise, $x.rR \ge r_2 > r_1$ so that $mw_1$ returns *nack* and $W_1$ aborts. Thus, $R_2$ returns the tuple written by a rmw-*write*procedure $mw'$ which is either $mw_1$ or some rmw-*write*procedure sequenced after $mw_1$. Let $r'$, $v'$ be the arguments passed to $mw'$. Then, $r' \ge r_1$, since otherwise, $x.wR \ge r_1 > r'$ so that the value of $x$ remains unchanged. Moreover, by the rank-uniqueness assumption, $r' = r_1$ implies that $mw' = mw_1$. Therefore, $\langle r, v\rangle = \langle r', v'\rangle$ and either $\langle r', v'\rangle = \langle r_1, v_1\rangle$, or

$r' > r_1$ as needed. □

**Lemma 3.** *The pseudocode in Figure 4.2 satisfies Non-Triviality.*

*Proof.* According to the pseudocode, a rr-*write* operation $W$ with rank $r$ aborts if the rmw-*write*() procedure $w$ called within $W$ returns *nack*. This happens if $w$ sees $x.rR > r$ or $x.wR \geq r$. This is only possible if some rmw-*write*() procedure with rank $r' \geq r$, or a rmw-*read*() procedure with rank $r' > r$ is sequenced before $w$. This could happen only as a result of some previously returned or concurrent rr-*read* (rr-*write*) with rank $r' > r$ ($r' \geq r$). By the rank-uniqueness assumption, no two rr-*write* operations are ever invoked with the same rank. Therefore, $W$ can abort only due to some previously returned or concurrent rr-*read* or rr-*write* with rank $r' > r$ as needed. □

**Lemma 4.** *The pseudocode in Figure 4.2 satisfies Liveness.*

*Proof.* Liveness trivially holds since both rr-*read* and rr-*write* always return something (i.e., the implementation is *wait-free*). □

We have proven the following theorem:

**Theorem 3.** *The pseudocode in Figure 4.2 is an implementation of a ranked register object.*

### 4.5.2 A fault-tolerant construction of a ranked register object for NR-Crash

In this section we present a wait-free implementation of a ranked register object from ranked register objects that may experience non-responsive crash faults. The register supports an unbounded number of clients. Our construction utilizes $n$ shared ranked register objects up to $\lfloor (n-1)/2 \rfloor$ of which can incur non-responsive crash. The pseudocode appears in Figure 4.3. This construction is also straight-forward: Reading and writing are both done at a majority of the ranked register objects. As for rr-*write*, if any of the ranked register objects which are accessed returns *abort*, the operation aborts.

**Lemma 5.** *The pseudocode in Figure 4.3 satisfies Safety.*

*Proof.* That a rr-*read* operation can only return a valid value that was actually used in a rr-*write* operation or $\langle r_0, \perp \rangle$ is obvious from the code. Now consider a rr-*write* operation $W_1 =$ rr-$write(\langle r_1, v_1 \rangle)_i$ that commits and let $R_2 =$ rr-$read(r_2)_j$, $r_2 > r_1$ be a rr-*read* operation which returns $\langle r, v \rangle$. Since both $W_1$ and $R_2$ access at least $\lceil (n+1)/2 \rceil$ ranked register objects, there exists a single register $rr_k$ accessed by both $W_1$ and $R_2$. Moreover, the Safety of $rr_k$

```
Shared: Ranked register objects $rr_i$, $1 \leq i \leq n$

Local: Multisets $S_1 \subseteq RVals$, $S_2 \subseteq \{commit, abort\}$.

rr-$read(r)$:
    $S_1 \leftarrow \emptyset$
    || invoke rr-$read(r)$ on $rr_i$ for each $i$;
        for each returned response $resp$, store $resp$ in $S_1$;
    **wait** until $|S_1| \geq \lceil (n+1)/2 \rceil$;
    $\langle r, v\rangle \leftarrow \langle r', v'\rangle : \langle r', v'\rangle \in S_1 \wedge r' = max_{\langle r'', v''\rangle \in S_1} r''$
    return $\langle r, v\rangle$


rr-$write(\langle r, v\rangle)$:
    $S_2 \leftarrow \emptyset$
    || invoke rr-$write(\langle r, v\rangle)$ on $rr_i$ for each $i$;
        for each returned response $resp$, store $resp$ in $S_2$;
    **wait** until $|S_2| \geq \lceil (n+1)/2 \rceil$;
    **if** ($abort \in S_2$)
        return $abort$
    return $commit$
```

Figure 4.3: A fault-tolerant ranked register object construction for NR-Crash

ensures that the tuple $\langle r', v'\rangle$ returned by $rr_k$.rr-$read(r_2)_i$ must satisfy $r' \geq r_1$. Since $R_2$ returns the tuple with maximum rank, $r \geq r' \geq r_1$ as needed. □

**Lemma 6.** *The pseudocode in Figure 4.3 satisfies Non-Triviality.*

*Proof.* According to the protocol, a rr-*write* operation $W =$ rr-$write(\langle r, v\rangle)_i$ aborts if there exists $k$ such that $rr_k$.rr-$write(\langle r, v\rangle)_i$ aborts. By the Non-Triviality of $rr_k$, this can happen only if some invocation $rr_k$.rr-$write(\langle r', v'\rangle)_j$ ($rr_k$.rr-$read(r')_j$) with $r' > r$ occur before or concurrently to $rr_k$.rr-$write(\langle r, v\rangle)_i$. This can only be the case if some rr-*write* or rr-*read* operation with rank $r'$ has been completed before or is concurrent to $W$. □

**Lemma 7.** *The pseudocode in Figure 4.3 satisfies Liveness.*

*Proof.* Each rr-*write* or rr-*read* operation is guaranteed to terminate since at most $\lceil (n+1)/2 \rceil$ ranked register objects are required to respond, no more than $\lfloor (n-1)/2 \rfloor$ ranked register objects can incur non-responsive crash, and each individual non-faulty ranked register object is wait-free. □

We have proven the following theorem:

**Theorem 4.** *The pseudocode in Figure 4.3 is a wait-free construction of a ranked register object out of $n$ ranked register objects such that at most $\lfloor (n-1)/2 \rfloor$ can incur non-responsive crash faults.*

## 4.6 Lower Bounds

In this section we show two lower bounds on memory complexity of constructing services from $n$-reader-$n$-writer registers. The first result shows that at least $\Omega(\sqrt{n})$ are required to implement a wait-free consensus in an asynchronous shared memory system with a failure detector of class $\Omega$. This lower bound is a simple corollary from [FHS98]. It justifies our use of read-modify-write objects in the wait-free consensus algorithm of Section 4.4 to support an unbounded number of clients with constant memory consumption.

The second result proves that at least $\Omega(n)$ $n$-reader-$n$-writer registers are necessary for implementing the ranked register object used by the consensus implementation of Section 4.4. This result justifies our use of read-modify-write objects for implementing a ranked register object accessible by unbounded number of clients. Our prove employs the register covering technique of Lynch and Burns [BL93]. Note that this memory bound is also tight because a ranked register can be implemented using $n$ 1-writer-$n$-reader registers using the technique similar to the Disk Paxos protocol of Gafni and Lamport [GL03].

### 4.6.1 The necessity of $\Omega(\sqrt{n})$ read-write registers for implementing a wait-free $\Omega$-based consensus among $n$ processes

We first show that any wait-free $\Omega$-based consensus algorithm for $n$ processes can be used to derive an asynchronous *obstruction-free* consensus algorithm. The obstruction freedom progress guarantee is defined as follows: Fix $x$ to be a shared object and $\sigma$ be a sequence of invocations and responses of $x$.

**Definition 3 (Obstruction Freedom).** *$\sigma$ satisfies* obstruction freedom *if every invocation by a correct process is complete unless it is concurrent to an incomplete invocation by a correct process. $x$ is* obstruction-free *if all its fair runs satisfy obstruction freedom.*

We now show how any wait-free consensus algorithm for an asynchronous shared memory model augmented with a failure detector of class $\Omega$ can be transformed to an asynchronous obstruction-free consensus algorithm. Let $\mathcal{A}$ be an automaton composed of process automata $P_1, \ldots, P_n$, $n$-reader-$n$-writer objects, $O_1, \ldots, O_m$, $m > 0$. We model a failure detector $\mathcal{L}$ as an abstract environment automaton whose inputs are of the form $query_i$, and outputs are $leader(j)_i$, where $1 \leq i, j \leq n$. Each process $P_i$, $1 \leq i \leq n$ interacts with $\mathcal{L}$ by means of the matching output actions $query_i$ and input actions $leader(j)_i$.

In order to obtain an asynchronous obstruction-free consensus implementation we first introduce a collection of automata $L_1, \ldots, L_n$, where each $L_i$ is implemented as follows:

```
States: $status \in \{idle, queried\}$, initially $idle$;

$query_i$:
    Effect:
        $status \leftarrow queried$;

$leader(j)_i$:
    Pre:
        $status = queried$;
        $j = i$;
    Eff:
        $status \leftarrow idle$;
```

We then obtain an automaton $A$ by adding automata $L_1, \ldots, L_n$ to $\mathcal{A}$ so that each process automaton $P_i$ of $\mathcal{A}$ is composed with $L_i$. We prove the following:

**Lemma 8.** *Suppose that $A \times \mathcal{L}$ is an implementation of a wait-free consensus object for all $\mathcal{L} \in \Omega$. Then, $A$ implements an obstruction-free consensus object.*

*Proof.* We first argue that $A$ satisfies agreement and validity. Let $\alpha$ be a finite execution of $A$. Since a failure detector of class $\Omega$ is allowed to exhibit an arbitrary behavior in a finite execution, for every finite execution $\alpha$ of $A$, there exists a failure detector $\mathcal{L}' \in \Omega$ such that $\alpha$ is an execution of $\mathcal{A} \times \mathcal{L}'$. Since $\mathcal{A} \times \mathcal{L}$ must satisfy agreement and validity for all $\mathcal{L} \in \Omega$, it must satisfy these two properties for $\mathcal{A} \times \mathcal{L}'$ as well. Hence, both agreement and validity hold in $\alpha$.

It remains to show that $A$ satisfies obstruction freedom. Suppose to the contrary. Let $\alpha$ be a fair run of $A$ such that there exists a correct process $i$ that runs alone after some finite prefix $\alpha'$ of $\alpha$. By assumption, process $i$ does not decide in $\alpha$. We construct an execution $\beta$ of $A$ as follows: The execution $\beta$ starts with all the events of $\alpha'$. It is then followed by $crash_j$ events for all processes $j \neq i$ that have not yet crashed in $\alpha'$. The remainder of $\beta$ consists of all the events that occur in $\alpha$ after $\alpha'$. Since failure detectors in $\Omega$ are allowed to exhibit an arbitrary behavior in finite runs, and because eventually they are required to permanently trust a single correct object, there exists a failure detector $\mathcal{L}' \in \Omega$ such that $\beta$ is a run of $\mathcal{A} \times \mathcal{L}'$. Since $\mathcal{A} \times \mathcal{L}$ satisfies wait freedom for all $\mathcal{L} \in \Omega$, $A \times \mathcal{L}'$ satisfies wait freedom as well. Hence, process $i$ must decide in $\beta$. However, $\alpha$ and $\beta$ are indistinguishable to process $i$. Consequently, since by assumption process $i$ does not decide in $\alpha$, it cannot decide in $\beta$ as well. A contradiction. □

We now re-state the lower bound of [FHS98]. This lower bound applies to the asynchronous consensus implementation satisfying the following weak termination guarantee:

**Definition 4 (Solo Termination [FHS98, JTT00]).** *An object $x$ is* solo-terminating *if every finite trace of $x$ can be extended into a fair trace $\sigma$ such that $\sigma$ satisfies obstruction freedom (see Definition 3 above).*

The lower bound of [FHS98] is as follows:

**Theorem 5 (Fich, Herlihy and Shavit [FHS98]).** *A solo-terminating implementation of $n$-process consensus requires $\Omega(\sqrt{n})$ $n$-reader-$n$-writer registers.*

The space complexity lower bound for a wait-free consensus in an asynchronous shared memory system with a failure detector of class $\Omega$ is a direct consequence of Theorem 5 and Lemma 8.

**Theorem 6.** *A wait-free implementation of $n$-process consensus in an asynchronous shared memory system augmented with a failure detector of class $\Omega$ requires $\Omega(\sqrt{n})$ $n$-reader-$n$-writer registers.*

*Proof.* By Lemma 8, there exists a shared memory preserving reduction from the obstruction-free asynchronous consensus in an asynchronous shared memory system to the wait-free $n$-process consensus in an asynchronous shared memory system augmented with a failure detector of class $\Omega$. By Theorem 5, a solo-terminating implementation of $n$-process consensus requires $\Omega(\sqrt{n})$ $n$-reader-$n$-writer registers. Since an obstruction-free consensus implementation is also solo-terminating, it requires at least as many registers. Hence, a wait-free implementation of $n$-process consensus in an asynchronous shared memory system augmented with a failure detector of class $\Omega$ requires $\Omega(\sqrt{n})$ $n$-reader-$n$-writer registers as well. □

### 4.6.2 The necessity of $\Omega(n)$ read-write registers for implementing a ranked register accessible by $n$ processes

Our construction of a fault tolerant ranked register object requires strong (read-modify-write) base objects. In this section we address the natural question of whether this strong memory model is necessary. We prove that a ranked register object cannot be implemented using a bounded number of atomic read/write registers (of unbounded size) in the presence of unbounded number of clients, even if clients are failure-free. The main result of this section is expressed in Theorem 7 below. It shows that any algorithm that implements the ranked register object specification in a shared memory system with $n$ processes must use at least $n$ atomic read/write registers. It then follows that if the number of processes is not bounded, the number of shared read/write registers needed to implement the ranked register object is also unbounded.

In order to prove this result, we utilize the technique of [BL93] to prove lower bounds on the number of atomic registers needed to solve mutual exclusion. Though the proof technique below is standard, it should be noted that there is no known direct reduction from mutual exclusion to a ranked register object, and hence, the results of [BL93] do not apply directly to the impossibility of constructing a ranked register object. In fact, we conjecture that the ranked register object is strictly weaker than the mutual exclusion problem, and hence, that no such reduction is possible.

We also note that it is possible to obtain the ranked register implementation space lower bound by applying the result of [JTT00], that shows that any randomized non-blocking linearizable implementation of a *perturbable* object type accessible by $n$ processes requires at least $n-1$ read-write registers[3] Note however, that this will result in a lower bound which is not as tight as ours. Also, proving an object type $T$ being perturbable requires reasoning about all possible linearizable implementations of an object of type $T$ which would obscure the otherwise simple and intuitive argument below.

We start with some definitions. We say that two system states $s$ and $s'$ are indistinguishable to process $i$, denoted $s \overset{i}{\sim} s'$, if the state of process $i$ and the values of all shared variables are the same in $s$ and $s'$. We say that process $i$ *covers* shared variable $x$ in system state $s$ if $i$ is about to write on $x$ in $s$.

**Lemma 9.** *Suppose that there exists an algorithm that implements a ranked register using only shared atomic read/write registers. Let $s$ be a reachable system state in which $r$ is the highest rank that appears in any operation. Then a* rr*-write operation $W =$* rr*-write$(\langle r', v' \rangle)_i$ by process $i$ with $r' > r$ must write some shared variable which is not covered in $s$.*

*Proof.* Assume in contradiction that no non-covered shared variable is written by $i$ in the course of $W$. We construct a system execution which violates the Safety property of the ranked register object as follows:

[3] The result in [JTT00] is even more general as it holds for perturbable type implementations from any combinations of objects in the set $\{resettableconsensus\} \cup \{historylessobjects\}$. In particular, read-write registers are historyless objects.

We first run from $s$ each process which covers some shared variable exactly one step so that they write the shared variables they cover. Let $s'$ be the resulting system state.

Next, we construct an execution fragment $\alpha_1$ starting in $s'$ and not involving $i$ by invoking a rr-*read*$(r'')$ operation $R$ at some process $j \neq i$ whose rank $r''$ satisfies $r'' > r'$. By the Liveness and the Safety properties of the ranked register object, $R$ must return a value written by some rr-*write* operation with rank at most $r$.

We now construct another execution fragment $\alpha_2$ which starts from $s$ as follows: We run $i$ solo until $W$ commits; since no higher rank appears in $s$, by the Non-Triviality property $W$ must indeed commit. By assumption, it writes only shared variables that are covered in $s$. From the resulting state, we run each process which covers some shared variable exactly one step so they overwrite everything written by $i$ in its solo run. Let $s''$ be the resulting state. Since $s'' \overset{j}{\sim} s'$ for all $j \neq i$, we can extend $\alpha_2$ by running $\alpha_1$ from $s''$.

By the Safety property of the ranked register object, the rr-*read* operation $R$ must return the value written by $W$ in this execution. However, it returns a value written by a rr-*write* operation with rank at most $r$ thus violating safety. A contradiction. □

We now set off to prove the lower bound. We use the following strategy: We first prove using Lemma 10 that with any algorithm implementing the ranked register object for $n \geq 1$ processes, it is possible to bring the system to a state where at least $n-1$ shared variables are covered while running only $n-1$ processes. In this state we invoke a rr-*write* operation whose rank is higher than the the rank of every operation invoked so far. Since this rr-*write* operation must commit (Non-Triviality), by Lemma 9, it must write to some shared variable which has not been covered yet. This implies that another shared variable is needed in addition to the $n-1$ covered ones.

**Lemma 10.** *Suppose that there exists an algorithm that implements a ranked register for $n \geq 1$ processes using only shared atomic read/write registers. Let $s$ be any reachable system state. Then for any $k$, $1 \leq k \leq n-1$, there exists a state $s_k$ which is reachable from $s$ using steps of processes $1 \ldots k$ only, such that at least $k$ distinct variables are covered in $s_k$.*

*Proof.* The proof is by induction on $k$.

Basis: $k = 1$. Let $s$ be any system state. We first run process 1 until it returns from the last operation invoked on 1, if any. This must happen due to the Liveness property of the ranked register object. Let $t$ be the resulting system state.

In $t$, we let process 1 invoke a rr-*write* operation $W$ whose rank is higher than the ranks of all operations invoked so far. By Non-Triviality, $W$ must commit. By Lemma 9, $W$ must write some shared variable which is not covered in state $s$. We then run 1 until it covers this variable. The resulting state $s_1$ satisfies the lemma requirements.

Inductive step: Suppose the lemma holds for $k$, where $1 \leq k \leq n-2$. Let us prove it for $k+1$. Using the induction hypothesis, we run $k$ processes from $s$ until the state $s_k$ is reached where at least $k$ distinct shared variables are covered. Starting in $s_k$, Starting in $t$, we run process $k+1$ until the last operation invoked on $k+1$ returns. This must happen due to Liveness. Let $t$ be the resulting state.

In $t$ we let process $k+1$ invoke a rr-*write* operation $W$ whose rank is higher than the ranks of all operations invoked so far. By Non-Triviality, $W$ must commit. Moreover, by Lemma 9, $W$ must write some shared variable which is not covered in $s_k$. So we run $k+1$ until it covers this shared variable. The resulting state $s_{k+1}$ satisfies the lemma requirements. $\square$

We are now ready to prove the main theorem:

**Theorem 7.** *If there exists an algorithm that implements a ranked register object for $n \geq 1$ processes, then it must use at least $n$ shared atomic read/write registers.*

*Proof.* Assume in contradiction that there exists an algorithm which implements a ranked register object for $n \geq 1$ processes using $n-1$ shared read/write registers.

Let $s$ be the initial system state. Note that there are no covered variables in $s$. We use the result of Lemma 10 and run $n-1$ processes from $s$ until the state $s_{n-1}$ is reached where the processes cover $n-1$ distinct shared variables. We then invoke a rr-*write* operation $W$ on process $n$ whose rank is higher than the ranks of all operations invoked so far. By Non-Triviality, $W$ must commit. By Lemma 9, $W$ must write some shared variable which is not covered in $s_{n-1}$. However, all $n-1$ shared variables are covered in $s_{n-1}$. A contradiction. $\square$

# Chapter 5

# Eventually Safe Leader Election in Shared Memory Model[1]

*Everything is Eventual...*
The title of a story by Stephen King

## 5.1 Introduction

In order to allow wait-free agreement to be solved, it is well known that the environment must be eventually synchronous for sufficiently long. Intuitively, this requisite enables a unique leader to be established and enforce a decision. In this chapter we focus on the problem of implementing such eventually-safe leader election from the bare shared memory environment, under an eventual partial synchrony assumption.

Our approach introduces as a building block the abstraction of an *eventual lease*. Informally, a lease is a shared object that supports a CONTEND operation, such that when CONTEND returns 'true' at any process, it does not return 'true' to any other process for a pre-designated period. The lease automatically expires after the designated time period. In addition, our lease supports a RENEW operation which allows a non-faulty leader to remain in leadership (indefinitely).

A lease is substantially different from a mutual-exclusion object: By its very nature, it becomes possible for other processes to recover an acquired lease regardless of the actions of the others, including the case that the process that held the lease has failed. Additionally, a lease may not be safe for an arbitrarily long period. Indeed, in an eventually (or intermittently) partially synchronous settings, any lease's pre-designated exclusion period entails no safety guarantee during periods of asynchrony. However, when the system stabilizes, all

[1]This chapter is based on the paper by Chockler and Malkhi [CM03].

previous (possibly simultaneous) leases expire, and safety is recovered by the very nature of leases. Thus, despite any transient periods of instability, leases guarantee that once a system becomes synchronous for sufficiently long, it will be possible for processes to acquire exclusive leases. Renewals also provide automatic recovery: Only one renewal emerges successfully after system stabilization, despite any unstable past periods, and despite the possible existence of multiple simultaneous lease holders before the stability.

Our leases achieve the following desirable properties.

**Recoverability:** Any acquired lease automatically expires, and thus any lease held by a faulty process is recoverable. By comparison, a vast amount of research centers around the mutual exclusion problem, which focuses on granting eternal locks to contending processes without recovery. In such systems, if a lock is acquired by a process that later fails, there is no way to recover it back. Recoverability realizes the vision of modern, scaling distributed systems such as Jini. The philosophy there taboos locking, and advocates putting automatic expiration dates on all data objects to prevent garbage from piling up.

**Uniformity:** Leases may have anonymous client processes whose number is unlimited and unknown. In contrast, known abstractions such as the failure detectors, including the $\Omega$ leader oracles of [CT96], are defined for a group of known members,

**Eventual-safety:** The system may go for arbitrarily long without stability, and hence, no safety. After the system becomes synchronous (and for as long as it stays in this state) each successful CONTEND or RENEW operation provides a pre-designated exclusive period to the contender, such that no other CONTEND/RENEW operation succeeds during this period. In this way, the system automatically re-stabilizes despite any transient asynchrony.

**Renewal:** An important feature of leases is their support of renewal by the current leader. This leads to efficient utilization of the lease by a leader who holds the lease and continues doing useful work. Even in case that multiple leases are held simultaneously during periods of instability, renewals guarantee that after stability, only a single leader succeeds in repeatedly renewing its lease.

Our lease immediately provides for eventually-safe leader election, which is at the heart of the shared memory version of the Paxos protocol [GL03]. The Paxos protocol guards the safety of decisions at all times, while the leader election oracle provides eventual progress. In this way, leases enable solving the Consensus problem we considered at the outset.

The lease object also has direct uses in storage area networks (SAN), in which clients access shared storage units directly over an Internet to provide high bandwidth and cost savings. First, a key service of a file system built for SANs is to manage leases that arbitrate access to shared files. Second, leases provide the bootstrapping of protocols like that of [GL03], in providing exclusive access to a dynamic directory listing that contains the names of clients.

Our leases provide uniform leader election, i.e., do not require any known bound on the number of client processes. Thus, coupled with the uniform shared-memory consensus protocol of Chapter 4, we obtain a uniform shared memory consensus protocol for partially synchronous systems.

This chapter provides the first implementation of leases from regular shared multi-reader/multi-writer read/write registers [SPW03]. The implementation builds largely on the timing-based simple mutual exclusion protocol of Fischer (see [Lam87]), but extends it in a number of important aspects. First, it handles process failures. Second, it adapts it to a shared memory model we employ, called the *Eventual Known Delay Timed* ($\Diamond$ND) model. This model considers remote memory operations to have non-zero duration. In addition to arbitrarily long period of asynchrony, the model captures both process failures and emulations of shared registers from fail-prone shared objects. Third, it provides a RENEW operation. Last, we build a leader-election module on top of leases, and demonstrate its applicability for solving the consensus problem.

In summary, the contribution of this chapter is in the specification of leases, in providing an implementation, and in demonstrating their uses.

## 5.2 System Model

In this chapter, we will assume a basic asynchronous shared memory model consisting of finite but a priori unknown universe of processes $p_1, p_2, \ldots$ communicating by means of a finite collection of reliable shared objects, $O_1, \ldots, O_n$. The objects constructed in this chapter utilize a single reliable wait-free multi-writer/multi-reader regular register[2]. Our definition of multi-writer regularity (see Section 5.2.1) follows the basic formalism of [SPW03]. In Section 5.2.2, we augment the basic model with necessary timeliness assumptions by adapting message-passing semi-synchronous models of [DLS88, CF99] to the shared memory environment.

[2]A fault-tolerant construction of a multi-writer/multi-reader regular register out of faulty memory objects can be found elsewhere (see e.g., [SPW03]).

### 5.2.1 The Basic Model

An *execution* of an object is a sequence of possibly interleaving invocations and responses. For an execution $\sigma$ and a process $p_i$, we denote by $\sigma|i$ the subsequence of $\sigma$ containing invocations and responses performed by $p_i$. An execution $\sigma$ is *admissible* if the following is satisfied: (1) Every invocation by a correct process in $\sigma$ has a matching response; and (2) For each process $p_i$, $\sigma|i$ consists of alternating invocations and matching responses beginning with an invocation. In the rest of this chapter, only admissible executions will be considered.

Given an execution $\sigma$, we denote by $ops(\sigma)$ (resp. $write(\sigma)$) the set of all operations (resp. all *write* operations) in $\sigma$; and for a *read* operation $r$ in $\sigma$, we denote by $writes_{\leftarrow r}$ the set of all *write* operations $w$ in $\sigma$ such that $w$ begins before $r$ ends in $\sigma$. The operations in $ops(\sigma)$ are partially ordered by a $\rightarrow_\sigma$ relation satisfying $o_1 \rightarrow_\sigma o_2$ iff $o_1$ ends before $o_2$ begins in $\sigma$. In the following, we will often omit the execution subscript from $\rightarrow$ if it is clear from the context.

Our definition of regularity for a multi-reader/multi-writer read/write shared object is similar to the **MWR2** condition of [SPW03]. It is as follows:

**Definition 5 (Regularity).** *An execution $\sigma$ satisfies regularity if there exists a permutation $\pi$ of all the operations in $ops(\sigma)$ such that for any read operation $r$, the projection $\pi_r$ of $\pi$ onto $writes_{\leftarrow r} \cup \{r\}$ satisfies:*

1. *$\pi_r$ is a legal sequence.*
2. *$\pi_r$ is consistent with the $\rightarrow$ relation on $ops(\sigma)$.*

*A read/write shared object is* regular *if all its executions satisfy regularity.*

### 5.2.2 The Augmented model

In the augmented model, each process is assumed to have access to a hardware clock with some predetermined granularity. We also assume that each process can suspend itself by executing a *delay* statement. Thus, a call to *delay*$(t)$ will cause the caller to suspend its execution for $t$ consecutive time units. The system is called *stable* over a time interval $[s, t]$, called a *stability period*, if the following holds during $[s, t]$: (1) The processes' clock drift with respect to the real-time is equal to a known real constant $\rho < 1$. For simplicity we assume that processes are eventually synchronous, i.e., $\rho = 0$ (it is easy to extend our results to clocks with $\rho \neq 0$); and (2) The time it takes for a correct process to complete its access to a shared memory object, i.e., to invoke an operation and receive a reply, is strictly less than a known positive integer $\delta$.

In the following, we will be interested mainly in properties exhibited by the system during stability periods. To simplify the presentation, we will consider a timed model, which we call an *Eventually Known Delay Timed* model, or $\Diamond$ND, with stability periods of infinite duration: i.e., we assume that for each run there exists a *global stabilization time (GST)* such that the system is stable forever after GST (i.e., during $[GST, \infty]$). In the remainder of the presentation, all properties and correctness proofs regard operations that start after GST.

We will also consider a special case of the $\Diamond$ND model, which we call a *Known Delay Timed* model, or ND, that requires each run to be stable right from the outset.

## 5.3 The Lease Implementation

A $\Delta$-*Lease* object supports a single operation CONTEND and is required to satisfy the following properties after GST:

**Property 5 (Safety).** *If a* CONTEND *operation $L$ by a process $p$ returns at time $t$, then there exists no* CONTEND *operation $L'$ by a process $q \neq p$, that returns within the interval $[t, t + \Delta]$.*

**Property 6 (Progress).** *If a correct process invokes a* CONTEND *operation at time $t$, then some* CONTEND *operation eventually returns after $t$.*

The $\Delta$-*Lease* object implementation appears in Figure 5.1. It utilizes a single shared multi-reader multi-writer regular register $x$.

We now prove that the implementation in Figure 5.1 satisfies the $\Delta$-Lease object properties.

We first note that due to the very nature of leases, the implementation cannot tell the difference between a faulty process and a correct process whose lease has expired. Therefore, throughout the proofs, we will restrict our attention to the runs where failures never occur. In addition, we make use of the following assumptions and notations. Let $L$ be a CONTEND operation. We denote the sequence of *read/write* operations by which $L$ terminates by:

$$L.r', \text{ (delay } \Delta + 5\delta), \; L.r'', \; L.w, \text{ (delay } 2\delta), \; L.r \quad .$$

That is, denote by $L.w$ the last *write* operation invoked during $L$ (i.e., the last time line 9 in Figure 5.1 is activated). Denote by $L.r$ the *read* operation that follows $L.w$ (on line 11), and by $L.r''$ the *read* operation that immediately precedes $L.w$ (invoked from line 6). Let $L.r'$ be the *read* operation during $L$ that precedes $L.r''$ (invoked either on line 1 or on line 11).

Finally, for the execution considered in all proofs, let $\pi$ be a serialization of the operations that upholds the regularity of $x$.

```
Shared: Regular register $x \in TS$;
Local: $x_1, x_2 \in TS$.

Process $i$:

CONTEND$_i$:
(1)  $x_2 \leftarrow read(x)$;
(2)  do
(3)      do
(4)          $x_1 \leftarrow x_2$;
(5)          $delay(\Delta + 5\delta)$;
             /* $\Delta + 6\delta$ for the $\Diamond$ND renewals */
(6)          $x_2 \leftarrow read(x)$;
(7)      until $x_1 = x_2$;
(8)      Generate a unique timestamp $ts$;
(9)      $write(x, ts)$;
(10)     $delay(2\delta)$;
(11)     $x_2 \leftarrow read(x)$;
(12) until $x_2 = ts$;
(13) return;
```

Figure 5.1: The $\Delta$-Lease Implementation.

**Lemma 11.** *Let $L_0$ be a* CONTEND *operation invoked by process $p$ that returns at time $t_0$. Denote $s_0 = t_0 + \Delta$ the expiration time of $L_0$. Then there exists no* CONTEND *operation $L$ such that $L.w$ appears in $\pi$ after $L_0.w$, and $L.r''$ is invoked before $s_0 + \delta$.*

*Proof.* Assume to the contrary, and let $L$ be a CONTEND operation such that $L.w$ is the first *write* in $\pi$ that breaks the conditions of the lemma.

Clearly, $L.w$ does not precede $L_0.r$ in $\pi_{L_0.r}$, for else $L_0.r$ cannot return the value written by $L_0.w$. Furthermore, since all *write* operations $w$ such that $w \to L_0.r$ must appear in $\pi_{L_0.r}$ before $L_0.r$, and because by assumption $L_0.w$ precedes $L.w$ in $\pi$, $L.w \not\to L_0.r$. Putting this together with the fact that the response of $L_0.w$ and the start of $L_0.r$ are separated by a $2\delta$ delay, we have $L_0.w \to L.r''$ (see Figure 5.2(a)). Hence, $L_0.w \in \pi_{L.r''}$.

Next, we show that $L_0.w$ is the last *write* preceding $L.r''$ in $\pi_{L.r''}$. Let $L' \neq L$ be a CONTEND operation such that $L'.w$ is between $L_0.w$ and $\pi_{L.r''}$ in $\pi_{L.r''}$. By assumption, $L'.r''$ must be invoked after $s_0 + \delta$. Since, by definition of $\pi_{L.r''}$, $L'.w$ must be invoked before $L.r''$

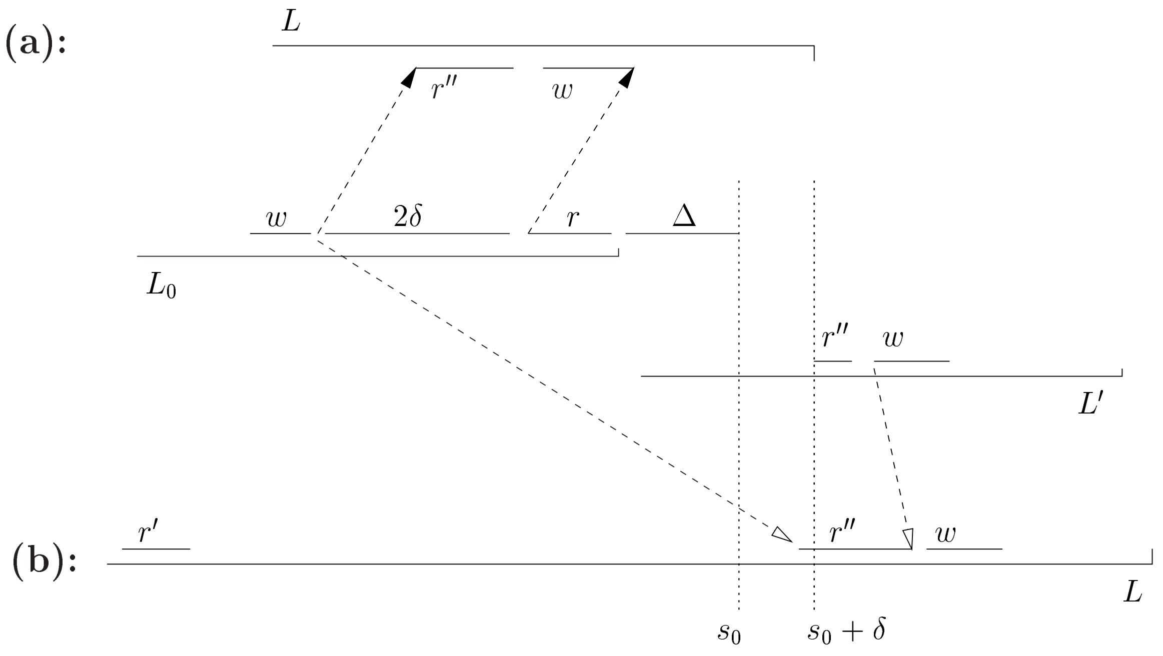

Figure 5.2: Possible placements of overlapping CONTEND operations $L_0$ and $L$.

returns, $L.r''$ returns after $s_0 + \delta$, as depicted in Figure 5.2(b). Since $L'.w$ is invoked after $s_0 + \delta$, and since by assumption, $L.r'$ finishes before $s_0 + \delta$, we get that $L.r' \to L'.w$. Putting this together with the assumption that $L'.w$ precedes $L.r''$ in $\pi_{L.r''}$, we obtain that $L.r'$ and $L.r''$ will return different values in which case the lease implementation implies that the *write* statement is not reached. Hence, $L.w$ could not have been invoked. Thus, $L_0.w$ is the last *write* preceding $L.r''$ in $\pi_{L.r''}$ implying that $L.r''$ returns the value written by $L_0.w$.

By construction, $L.r''$ is preceded by a $5\delta + \Delta$ delay preceded by another *read* operation $L.r'$ such that the timestamp values returned by these two *read*'s are identical. However, it is easy to see that $L_0.w$ is contained in full between these two reads. Indeed, we already know that $L_0.w \to L.r''$. We now show that $L.r' \to L_0.w$. Indeed, the earliest time that $L_0.w$ can be invoked is $s_0 - \Delta - 4\delta$. Since by assumption $L.r''$ is invoked before $s_0 + \delta$, $L.r'$ **returns** before $s_0 + \delta - (\Delta + 5\delta) = s_0 - \Delta - 4\delta$ (see Figure 5.2(b)). Therefore, $L.r' \to L_0.w$. Thus, regularity of $x$ and the timestamp uniqueness imply that $L.r'$ and $L.r''$ return different timestamps in which case the lease implementation implies that the *write* statement is not reached. Hence, $L.w$ could not have been invoked. A contradiction. □

We are now ready to prove Safety.

**Lemma 12 (Safety).** *The implementation in Figure 5.1 satisfies Property 5.*

*Proof.* Let $L$ be a CONTEND operation by process $p$ that returns at time $t$. Denote $s = t + \Delta$. Suppose to the contrary that another CONTEND operation $L'$ returns at time $t'$ within the interval $[t, s]$.

First, $L'.r''$ must be invoked before $s+\delta$. By Lemma 11, putting $L_0 = L$ we get that $L.w$ does not precede $L'.w$ in $\pi$. Second, $L.r''$ must be invoked before $t'$, and a fortiori, before $t' + \Delta + \delta$. Applying Lemma 11 again, with $L_0 = L'$, we get that $L.w'$ does not precede $L.w$ in $\pi$. A contradiction. $\square$

We now turn our attention to proving Progress. We first prove the following technical fact.

**Lemma 13.** *Let $q$ be a process that performs an operation $w_1 =$ write that returns at time $t$. If no process returns from a* CONTEND *operation after $t$, then for each $s > t$, the interval $[s, s + 5\delta]$ contains a complete write invocation (i.e., from its invocation to its response).*

*Proof.* Suppose to the contrary. By assumption, no *write* operation is invoked between $s$ and $s+4\delta$. Let $W$ be the last *write* invoked before $s$, or possibly the set of concurrent, latest *write*s invoked before $s$. Formally, $W$ is the set of all $w$ such that (1) $w$ is invoked before $s$; and (2) for any write $w'$ invoked by $s + 4\delta$, $w \not\to w'$. $W$ is not empty because $w_1$ starts before $s$, and no write is invoked in the interval $[s, s + 4\delta]$.

Let $w \in W$, and let $r =$ *read* be the corresponding read operation, invoked by the same process $2\delta$ after $w$. We claim that (i) $W \to r$, and (ii) there does not exist any *write* event $\omega$ in $\pi_r$ that follows $W$ in $\pi$ such that $W \to \omega$ and $\omega$ is invoked before $r$ returns.

To see that $(i)$ holds, let $w' \in W$. Since $w \not\to w'$, we have that $w'$ terminates at most $\delta$ after $w$; since $r$ starts $2\delta$ after $w$'s termination, $w' \to r$. To see $(ii)$, first note that if $W \to \omega$, then by definition $\omega$ cannot be invoked before $s$. Second, by assumption, no *write* is invoked between $s$ and $s+4\delta$, but $r$ terminates by $s+4\delta$ at the latest. So $\omega$ cannot be invoked before $r$ returns, and hence is not in $\pi_r$.

Hence, by the regularity of $x$, all *read*ś corresponding to *write*ś in $W$ must return the value of the last *write* in $\pi$ from $W$. The *read* corresponding to this *write* then sees $x$ unchanged, and its initiator is allowed to obtain the lease. A contradiction. $\square$

**Lemma 14 (Progress).** *The implementation in Figure 5.1 satisfies Property 6.*

*Proof.* Let $p$ be a correct process that invokes a CONTEND operation at $t$. Suppose for contradiction that no CONTEND operation returns after $t$.

First, eventually some process, say $q_1$, invokes an operation $w_1 =$ *write*. This is due to the fact that the wait-loop at the start of the CONTENDalgorithm (lines 5.13–7) terminates when no *write*ś are performed.

By Lemma 13, if there is no successful CONTEND after $w_1$ is invoked, then every instance of the loop by $q_1$ observes at least one new written value, so $q_1$ does not perform any further *write*ś. Let an operation $w_2 =$ *write* by $q_2$ be observed by $q_1$. Again, so long as there is no successful CONTEND, by Lemma 13, $q_2$ performs no further *write*ś. And so on.

Since the number of processes is finite, eventually all processes are in their wait loop and no process writes. This is a contradiction. □

## 5.4 Lease renewals

In many situations, it is important to enable the current lease holder to renew its lease without contention. For example, this is the case when a lease holder requires more time to complete an operation than the alloted period. Another example is the use of leases to obtain a leader, in which case we wish the leader to perpetuate so long as it is alive.

In this section we address lease renewals. We consider two renewal types: The first one is suitable for the ND model, and is extremely efficient. The second one works in the $\Diamond$ND model, and guarantees stabilization of renewal: Only one renewal emerges successfully after GST, despite any unstable past periods, and despite the possible existence of multiple simultaneous lease holders before GST. The $\Diamond$ND renewal protocol is somewhat more costly.

For the $\Diamond$ND model, the renewal strategy introduces a slightly modified interface with the RENEW operation returning a Boolean value indicating whether the renewal was successful or not. (For the ND model, a return from a renew operation is always considered successful).

We say that a process *renews* its lease if it calls a *renew* operation as soon as its previously granted lease of $\Delta$ time units expires. The $\Delta$-*Lease* implementation with renewals is required to satisfy the following properties:

**Property 7 (Renewal Safety).** *1. If a lease contend or renew operation $L$ returns successfully at a process $p$ at time $t$, then for any lease operation $L'$ (contend or renew) invoked by a process $q \neq p$, $L'$ does not return successfully in the interval $[t, t+\Delta]$.*

*2. Assume that a renewal operation $L$ by a process $p$ is invoked at time $t_1$ and returns successfully at time $t_2$. Then there exists no successful contend or renewal operation $L'$ by a process $q \neq p$ that returns during the interval $[t_1, t_2]$.*

**Property 8 (Renewal Liveness).** *If a lease contend or renew operation returns successfully at a correct process $p$ at time $t$, and process $p$ invokes renew at time $t$, then the renew succeeds.*

### 5.4.1 ND renewal

The renewal implementation in the ND model is extremely simple: A process whose previously granted lease expires can renew it for another $\Delta$ time units by simply executing lines 8–9 of the $\Delta$-*Lease* implementation in Figure 5.1. More precisely, we define the *renew* operation as follows:

```
renew:
    Generate a unique timestamp $ts$;
    $write(x, ts)$;
```

We now prove the correctness of the ND renewal scheme. Since liveness trivially holds, we are only left with proving safety.

**Lemma 15.** *Consider a sequence $\ell = L_0 rn_1 rn_2 \ldots rn_k$ of lease operations by process $p$. Suppose that $L_0$ is a successful* CONTEND *operation that returns at time $t_0$, and $rn_i$ is a successful* RENEW *operation that returns at time $t_i$. Then there exists no* CONTEND *operation $L$ by process $q \neq p$ such that $L.w$ is invoked within the interval $[t_0, t_k + \Delta + 2\delta]$.*

*Proof.* By induction on length of $\ell$. For the base case, let $\ell = L_0 rn_1$. Suppose to the contrary that there exists a CONTEND operation $L$ such that $L.w$ is invoked within $[t_0, t_1 + \Delta + 2\delta]$. First, note that $L_0.w \to L.w$, and therefore, $L_0.w$ precedes $L.w$ in $\pi$. Therefore, by Lemma 11, $L.r''$ must be invoked after $t_0 + \Delta + \delta$. Since $rn_1.w$ is invoked at $t_0 + \Delta$, it must return by $t_0 + \Delta + \delta$, and therefore, $rn_1.w \to L.r''$. Since $L.r''$ is invoked before $t_1 + \Delta + 2\delta$, $L.r'$ returns before $t_1 + \Delta + 2\delta - (\Delta + 5\delta) = t_1 - 3\delta$. Since $rn_1.w$ must be invoked at $t_1 - \delta$ the earliest, $L.r' \to rn_1.w$. Therefore, by regularity of $x$ and timestamp uniqueness, $L.r'$ and $L.r''$ will return different values violating the necessary condition for the *write* statement of the CONTEND implementation to be reached. Hence, $L.w$ cannot be invoked. A contradiction.

Assume that the result holds for all sequences $\ell$ of length $k-1$, and consider a sequence $\ell' = \ell\ rn_k$. Assume to the contrary. By the inductive assumption, $L.w$ must be invoked after $t_{(k-1)} + \Delta + 2\delta$. Therefore, $rn_k.w \to L.r''$. On the other hand, $L.r''$ must be invoked before $t_k + \Delta + 2\delta$. Therefore, $L.r'$ must return before $t_k - 3\delta$. Since the earliest time $rn_k.w$ can be invoked is $t_k - \delta$, $L.r' \to L.w$. Therefore, by regularity of $x$ and timestamp uniqueness, $L.r'$ and $L.r''$ will return different values violating the necessary condition for the *write* statement of the CONTEND implementation to be reached. Hence, $L.w$ cannot be invoked. A contradiction. ☐

**Lemma 16.** *Suppose that a process $p$ returns from a* RENEW *operation $rn$ at time $t$. Then, there exists no process $q \neq p$ whose* RENEW *operation $rn'$ returns within the interval $[t, t+\Delta]$.*

*Proof.* Suppose to the contrary that $rn'$ returns at time $t'$ within the interval $[t, t+\Delta]$. Note that both $p$ and $q$ must have been invoked contend operations $L$ and $L'$ in the past to acquire their initial leases. Suppose that $L$ and $L'$ return at times $c < t$ and $c' < t'$ respectively.

Assume, w.l.o.g, that $c < c'$. By Lemma 15, putting $t_0 = c$ and $t_k = t + \Delta$, and because $t' \leq t + \Delta$, we get that the lease period of $L'$ overlaps with $[t_0, s_k]$. A contradiction. $\square$

The following lemma follows immediately from Lemma 15 and Lemma 16.

**Lemma 17 (ND Renewal Safety).** *The ND renewal implementation satisfies Property 7 (Renewal Safety).*

We proved the following:

**Theorem 8 (ND Renewal Safety and Liveness).** *The ND renewal implementation satisfies Property 7 (Renewal Safety) and Property 8.*

### 5.4.2 $\Diamond$ND renewal

The RENEW operation implementation for the $\Diamond$ND model is shown in Figure 5.3. For simplicity, we require that timestamps consist of two fields: the process id and a monotonically increasing counter.

```
renew:
(1)  $x_1 \leftarrow read(x)$;
(2)  if $(x_1.id \neq ts.id)$ then
(3)      return $false$;
(4)  $ts.counter \leftarrow ts.counter + 1$;
(5)  $write(x, ts)$;
(6)  $delay(2\delta)$;
(7)  $x_1 \leftarrow read(x)$;
(8)  if $(x_1 = ts)$ then
(9)      return $true$;
(10) else
(11)     return $false$;
```

Figure 5.3: $\Diamond$ND Renew Implementation.

Throughout the proof of correctness of the $\Diamond$ND renewal scheme, we make use of the following notation. Let $L$ be a CONTEND or RENEW operation. As in the previous section, we denote the sequence of *read/write* operations by which $L$ terminates by:

(in CONTEND only: $L.r'$, delay $\Delta + 6\delta$), $L.r''$, $L.w$, (delay $2\delta$), $L.r$ .

That is, $L.w$ is the last *write* operation invoked within $L$, and $L.r''$, $L.r$ and the read operations immediately preceding and following $L.w$, respectively. If $L$ is a CONTEND operation, then in addition, the read operation preceding $L.r''$ is denoted $L.r'$.

Finally, for the execution considered in all proofs, let $\pi$ be a serialization of the operations that upholds the regularity of $x$.

**Lemma 18.** *Let $L_0$ be a lease operation (contend or renew) invoked by process p that returns successfully at time $t_0$. Denote $s_0 = t_0 + \Delta$ the expiration time of $L_0$. Then there exists no write operation $w$ in $\pi$ after $L_0.w$, such that $w$ is invoked before $s_0 + \delta$.*

*Proof.* Assume to the contrary, and let $L.w$ be the first *write* in $\pi$ that breaks the lemma.

Clearly, $L.w$ does not precede $L_0.r$ in $\pi_{L_0.r}$, for else $L_0.r$ cannot return the value written by $L_0.w$. Furthermore, since all *write* operations $w$ such that $w \rightarrow L_0.r$ must appear in $\pi_{L_0.r}$ before $L_0.r$, and because by assumption $L_0.w$ precedes $L.w$ in $\pi$, $L.w \not\rightarrow L_0.r$. Putting this together with the fact that the response of $L_0.w$ and the start of $L_0.r$ are separated by a $2\delta$ delay, we have $L_0.w \rightarrow L.r''$ (see Figure 5.4). Hence, $L_0.w \in \pi_{L.r''}$.

Furthermore, by assumption $L.w$ is the first *write* such that (1) $L.w$ follows $L_0.w$ in $\pi$; and (2) $L.w$ is invoked before $s_0 + \delta$. Since $L_0.w \in \pi_{L.r''}$ any write $w \neq L.w$ that follows $L_0.w \in \pi_{L.r''}$ must be invoked after $s_0 + \delta$. Since, by definition of $\pi_{L.r''}$, $w$ must be invoked before $L.r''$ terminates, $L.r''$ terminates after $s_0 + \delta$. Consequently, $L.w$ would be invoked after $s_0 + \delta$ contradicting the assumption. Since $L.w \notin \pi_{L.r''}$, the only remaining possibility is that $L_0.w$ is the last *write* in $\pi_{L.r''}$, and so $L.r''$ returns the value of $L_0.w$.

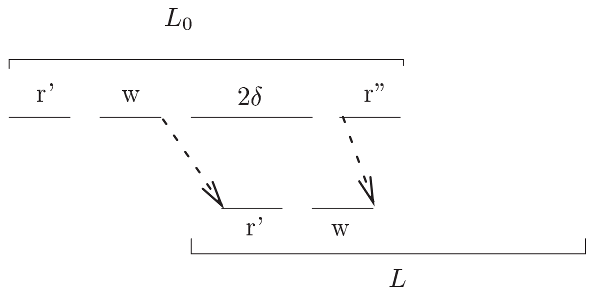

Figure 5.4: Overlapping renewals.

Next, we consider the case that $L$ is a CONTEND operation separately from the case that it is a RENEW operation. First, consider that $L$ is a RENEW operation. Then the analysis above shows that $L.r''$ returns the timestamp written in $L_0.w$, hence $L$ is unsuccessful.

Second, assume that $L$ is a CONTEND operation. Here, $L.r''$ is preceded by a $6\delta + \Delta$ delay preceded by another *read* operation $L.r'$: and the timestamp values returned by these two *read*'s are identical. However, it is easy to see that $L_0.w$ is contained in full between these two reads. We already know that $L_0.w \rightarrow L.r''$. We now show that $L.r' \rightarrow L_0.w$.

Indeed, the earliest time that $L_0.w$ can be invoked is $s_0 - \Delta - 4\delta$. Since by assumption $L.w$ is invoked before $s_0 + \delta$, $L.r'$ is invoked before $s_0 + \delta - (\Delta + 6\delta) = s_0 - \Delta - 5\delta$. Therefore, $L.r' \to L_0.w$. Thus, regularity of $x$ and the timestamp uniqueness imply that $L.r'$ and $L.r''$ return different timestamps in which case the lease implementation implies that the *write* statement is not reached. Hence, $L.w$ could not have been invoked. A contradiction. $\square$

We are now ready to prove Safety:

**Lemma 19 (Property 7.1).** *Assume that a lease operation $L$ (*CONTEND *or* RENEW*) by process $p$ returns successfully at time $t$. Let $s = t + \Delta$. Then there exists no successful* CONTEND *or* RENEW *operation $L'$ by a process $q \neq p$ that returns during the interval $[t, s]$.*

*Proof.* Suppose to the contrary that $L'$ returns successfully at time $t'$ within the interval $[t, s]$. First, $L'.w$ must be invoked before $s + \delta$. By Lemma 18, putting $L_0 = L$ we get that $L.w$ does not precede $L'.w$ in $\pi$. Second, $L.w$ must be invoked before $t'$, and a fortiori, before $t' + \Delta + \delta$. Applying Lemma 18 again, with $L_0 = L'$, we get that $L.w'$ does not precede $L.w$ in $\pi$. A contradiction. $\square$

**Lemma 20 (Property 7.2).** *Assume that a* RENEW *operation $L$ by a process $p$ is invoked at time $t_1$ and returns successfully at time $t_2$. Then there exists no successful* CONTEND *or* RENEW *operation $L'$ by a process $q \neq p$ that returns during the interval $[t_1, t_2]$.*

*Proof.* Suppose to the contrary that $L'$ returns at a time $t'$ within the interval $[t_1, t_2]$. First, $L'.w$ must be invoked before $s + \delta$. By Lemma 18, putting $L_0 = L$ we get that $L'.w$ must precede $L.w$ in $\pi$. Furthermore, applying Lemma 18 again with $L_0 = L'$, we get that $L.w$ must be invoked after $t' + \Delta + \delta$. Therefore, $L'.w \to L.r''$ so that $L'.w \in \pi_{L.r''}$, and $L'.w$ precedes $L.r''$ in $\pi_{L.r''}$.

First, suppose that $L.w$ is the first *write* operation by $p$ in $\pi$ after $L'.w$. Hence, there is no *write* operation by $p$ in $\pi_{L.r''}$ following $L'.w$. Then by regularity of $x$, and because $L$ is a RENEW operation, $L.r''$ returns a timestamp written by a process $q \neq p$, contradicting to the fact that $L$ is successful.

Next, suppose that there exists a *write* operation $L''.w$ by $p$ in $\pi_{L.r''}$ that follows $L'.w$. Since $L$ is a RENEW operation, $L''$ must be the successful lease (RENEW or CONTEND) operation immediately preceding $L$. Applying Lemma 18 with $L_0 = L'$, we get that $L''.w$ must be invoked after $t' + \Delta + \delta$ implying that $L$ starts after $t' + \Delta + \delta$ (i.e., $t_1 > t' + \Delta + \delta$). $\square$

We proven the following

**Theorem 9 (Renewal Safety).** *$\Diamond$ND renew implementation satisfies Property 7 (Renewal Safety).*

Finally, we prove Liveness:

**Lemma 21.** *Assume that a correct process $p$ obtains the lease in a* CONTEND *or* RENEW *operation $L$ at time $t$. Then, a* RENEW *operation $rn$ invoked by $p$ at $s = t + \Delta$, returns successfully.*

*Proof.* For $rn$ to be successful, first $rn.r''$ must return the timestamp written by $L.w$. This holds by the fact that $L.r$ returns the value of $L.w$, and by Lemma 18, since no other *write* operation that follows $L.w$ in $\pi$ is invoked before $s + \Delta + \delta$.

Second, $rn.r$ needs to return the value written by $rn.w$. Suppose to the contrary that some lease operation $L'$ overwrites $rn.w$. Let $L'.w$ be the first *write* in $\pi$ by process $q \neq p$ that follows $L.w$ and precedes $rn.r$ in $\pi_{rn.r}$.

By Lemma 18, $L'.w$ is invoked after $s + \delta$. Hence, $L.w \rightarrow L'.r''$. Since $L'.w$ is the first *write* to follow $L.w$, and since $L'.r'' \rightarrow L'.w$, we have that $L'.r''$ returns the timestamp written by $p$ in $L.w$. By construction, this occurs only if $L'$ is a CONTEND (not RENEW) operation. Still, for $L'.w$ to be invoked, $L'.r'$ and $L'.r''$ must return the same timestamp. We now show this is impossible.

We already know that $L.w \rightarrow L'.r''$. By construction, $L'.r''$ follows a delay of $\Delta + 6\delta$ after the termination of $L'.r'$. If $L'.r''$ is invoked no later than $s + 2\delta$, then $L'.r'$ terminates by $s - \Delta - 4\delta$. Since the earliest that $L.w$ is invoked is $t - 4\delta$, we have $L'.r' \rightarrow L.w$. We get that $L.w$ is a *write* that occurs completely between $L'.r'$ and $L'.r''$, and so they must return different timestamps.

We are left with the possibility that $L'.r''$ is invoked after $s + 2\delta$. Because $L'.w$ precedes $rn.r$ in $\pi_{rn.r}$, the latest that $L'.r''$ may be invoked is $s + 5\delta$. Hence, $L'.r'$ terminates by $s - \delta$. We now get that $rn.w$ is a *write* that occurs completely between $L'.r'$ and $L'.r''$, and so they return different timestamps.

Hence, $L.r'$ and $L'.r''$ must see different values, in contradiction to the assumption that $L'.w$ is invoked after $L'.r''$. Hence, $rn.r$ returns the same value as $rn.w$, and the renewal succeeds. □

## 5.5 Leader Election

In this section we show the lease-based implementation of a Boolean leader oracle $\mathcal{L}$, that is required by the universal service replication of Chapter 4 and by the Consensus algorithm of [GL03]. Recall that $\mathcal{L}$ is a failure detector of class $\Phi$ such that the output of a local failure detector module $\mathcal{L}_i$ at each process $i$ is a Boolean value satisfying the following property eventually:

```
Shared $\Delta$-Lease object $L$;
Local Boolean $leader$;

(1) forever do
(2)     $leader \leftarrow false$;
(3)     $L.lease()$;
(4)     $leader \leftarrow true$;
(5)     $delay(\Delta)$;
(6)     while ($L.renew()$) do
(7)         $delay(\Delta)$;
(8)     od
(9) od

When queried:
    return $leader$;
```

Figure 5.5: The Lease-based Leader Oracle implementation

**Property 9 (Unique Leader).** *There exists a correct process $i$ such that $\mathcal{L}_i$ permanently outputs true, and for each process $j \neq i$, $\mathcal{L}_j$ permanently outputs $false$.*

The lease based implementation of $\mathcal{L}$ appears in Figure 5.5. A complete Consensus algorithm based on $\mathcal{L}$ appears in Chapter 4 and [CM02].

The following theorem establishes the correctness of the leader oracle implementation in the $\Diamond$ND model.

**Theorem 10.** *The pseudocode in Figure 5.5 eventually satisfies Property 9 in the $\Diamond$ND model.*

*Proof.* Since the leader election code is executed in an infinite loop, eventually some CONTEND or RENEW operation will be invoked after GST. Moreover, there exists $T \geq GST$ such that for every lease operation $L$ invoked before GST, either the $L$'s lease period has already expired, or $L =$ CONTEND and $L$ never returns after $T$. Note that this time must exist since only a finite number of processes may invoke a lease operation before GST. By the lease liveness, once a correct process $p$ invokes a lease operation $L$ (CONTEND or RENEW) after $T$, $L$ as well as all the subsequent RENEW operations must return successfully. Let $T'$ be the time at which $L$ returns. By the lease safety, $p$ will be the only lease holder after $T'$. $\square$

# Part II

# Tolerating Malicious Failures

# Chapter 6

# Inherent Cost of Optimal Resilience in the Presence of Malicious Storage[1]

We proceed by extending our study of fault-tolerance in storage-centric systems to the NR-arbitrary failure model: i.e., an asynchronous shared memory system with multiple processes accessing fault-prone shared memory objects [AGMT95, JCT98]. In this chapter we show that achieving optimal resilience threshold of $t < n/3$ failures in the NR-arbitrary model incurs a high cost in terms of the communication complexity. In the next chapter, we show a Paxos-like agreement protocol that avoids the communication overhead by lowering the resilience threshold to $t < n/5$ faulty objects.

## 6.1 Introduction

In this chapter we show for the first time, how to emulate a wait-free shared register in this model using a collection of $3t + 1$ fault-prone base objects. We also show that emulations with this level of resilience require invoking substantially more operations on the base objects than constructions that have a lower resilience level.

The goal of this chapter is to investigate the costs and tradeoffs involved in designing reliable solutions for scalable decentralized storage-centric systems storage units prone to NR-arbitrary failures. This focus mandates that we regard the system as asynchronous. It also implies that servers storing memory objects (or disks in a large disk farm) can stop responding. Furthermore, we consider arbitrary corruption (Byzantine failures) of memory (at either the servers or the disks), since the protection of highly decentralized services is hard. Hence, we adopt the NR-Arbitrary failure model [JCT98]. We assume that a large number of ephemeral processes (clients) may access the memory objects, and hence, authenticating all

[1]This chapter is based on the paper by Chockler, Keidar and Malkhi [CKM03].

the stored data is not feasible. Finally, when processes access their information over a wide interconnect, temporary disconnections and process crashes are bound to happen. Hence, we assume that processes may fail to reach completion of operations they initiate, and we strive to devise *wait-free* solutions.

Previous work that constructed wait-free objects in this failure model used $4t+1$ [MR00], $5t+1$ [JCT98], or even $6t+1$ [CMR01] base objects. A natural question to ask is whether the resilience threshold $t < n/4$ (where $n$ is the number of fault-prone objects) is tight in this model. Intuitively, one would hope for a resilience bound of $t < n/3$. (It is easy to see that in this model it is impossible to obtain better resilience than $t < n/3$ [MAD02]). Several previous works have addressed this question, and have achieved better resilience by weakening the model in different ways – by adding synchrony [Baz00]; by storing authenticated (signed) self-verifying data at the servers [MR00, MAD02]; or by assuming that clients never fail and providing solutions that may block indefinitely if clients do fail [MAD02, ABO03]. However, $t < n/4$ is the best resilience threshold previously achieved for wait-free constructions in the model considered herein.

In contrast, the literature is abundant with message-passing consensus algorithms that are resilient to Byzantine failures of less than a third of the processes. Therefore, an appealing way to go about searching for a more resilient solution would be to try and adapt the techniques used in those algorithms to our model. Two basic techniques are used in achieving this threshold in consensus algorithms: authentication (using signatures) of all transmitted data (e.g., [DLS88, CL02]) or echoing (e.g., [BT85, DLS88]). Authentication can indeed be used to construct memory objects that are resilient to $t < n/3$ failures (see [MR00]). However, our model does not incorporate a signature scheme as needed for authentication. The second approach is to have each process echo all the values it receives to all the processes (e.g., see [DLS88]). Unfortunately, echoing cannot help us address the challenge we have set out to solve in this chapter. Indeed, if a correct process can correctly echo information to all other processes, this is essentially the same as having a wait-free register through which the process conveys the information to the other processes. And implementing such a register from fault-prone storage is exactly what we seek to do in this chapter.

Having ruled out the use of standard techniques to improve the resilience threshold, we proceeded to examine whether there are any inherent limitation that prevent algorithms in our model from achieving better resilience. We observe that existing algorithms for fault-prone shared memory models (e.g., [MR00, MAD02, Baz00, JCT98, ABO03]) implement (emulate) READ and WRITE operations in a single round; that is, they invoke one read or write operation on each base object. In Section 6.4.1, we prove that if $t \geq n/4$, then it is impossible to emulate the WRITE operations of a wait-free register by invoking a single round of operations on the base objects. Our proof applies to binary single-writer single-reader *safe* *registers*[2]; the weakest meaningful register type [Lam86].

Moreover, in Section 6.4.2, we show that if $n = 3t + 1$, then any algorithm in which the reader does not modify the base objects' states may need to invoke as many as $t + 1$ rounds of read operations on base objects in order to emulate a single READ operation of a single-writer single-reader safe register. More generally, for any $0 \leq f \leq t$, there is a run in which $f$ objects are Byzantine faulty in which the algorithm invokes $min(t+1, f+2)$ rounds of base object operations.

We now explain the challenge in working with a resilience of $n = 3t + 1$. Traditionally, in asynchronous algorithms, one waits for at most $n - t$ responses to each request. Since $t$ objects may be non-responsive, waiting for more objects may violate liveness. Thus, an emulated WRITE(v) operation returns once the lower-level write operations on $n - t$ base objects return. But of the $n - t$ base objects that return, $t$ may be actually faulty, whereas the $t$ that have not responded may be simply slow. In this case, only $n - 2t = t + 1$ correct base objects have stored the value V. If a READ operation is invoked after WRITE(v) returns, and no further WRITE operations are invoked, then safety semantics mandates that the READ return V. The READ operation probes the base objects, and waits for responses from $n - t$ of them (again, in order to ensure liveness). The set of $n - t$ responders may overlap the set of $t + 1$ correct objects that have V by as little as one object. Since data is not authenticated, the reader has no way of distinguishing a value V that is returned by $k < t+1$ correct objects from an arbitrary value $v'$ concocted by $k$ corrupt objects.

In order to overcome this difficulty, we observe that in some situations, after $n - t$ responses are received, waiting for additional responses will not violate liveness. To illustrate the crux of our technique, consider the example above, where there are no WRITE operations overlapping the READ. If the reader sees only $k$ instances of V, then there are two possible cases: Either (1) V is indeed a correct value, in which case there must be $t+1-k$ additional correct objects that store V, whose responses were not collected by the reader; or (2) V is forged, in which case $k$ of the responses are from faulty objects, and there are $k$ correct objects whose responses were not collected by the reader. In either case, a reader can wait for more responses until each value V either occurs in $t + 1$ responses, or does not appear in $2t + 1$ responses. Unfortunately, when there may be WRITE operations overlapping the READ, matters become significantly more subtle. This simple solution does not work, since the WRITE of V might be in progress, in which case neither condition will be met, and hence waiting for more replies would violate liveness. Our algorithm is therefore more elaborate.

In Section 6.3 we give a construction of a wait-free single-writer multi-reader safe register from $3t + 1$ registers, $t$ of which can suffer NR-Arbitrary faults. Our algorithm is *optimal*;

[2]A safe register guarantees that read operations that do not overlap any write operation return the correct (most recently written) value; a read operation that overlaps a write may return an arbitrary value.

the number of rounds of operations it invokes matches the lower bounds of Sections 6.4.1 and 6.4.2. That is, our algorithm emulates WRITE operations using two rounds of base object write operations, and it emulates READ operations in $min(t+1, f+2)$ rounds of base objects read operations in runs with $f$ Byzantine failures. Moreover, our algorithm has the desirable property that in synchronous runs, as well as in runs with benign failures only, it always terminates after at most two rounds of base object invocations, regardless of the number of failures.

We chose to present an emulation of a single-writer multi-reader safe register, since this is the most basic building block that can be used for constructing various object types. E.g., there are well-known constructions of regular and atomic registers from safe ones [Lam86]. These, in turn, can be used to construct higher level objects; e.g., consensus can be solved with regular registers if the system is augmented with an oracle failure detector [LH94].

## 6.2 System Model

We consider an asynchronous shared memory system consisting of processes interacting with each other by means of a finite collection of shared objects, $O_1, \ldots, O_n$. Processes may fail by stopping (crashing). The implementation should be *wait-free* in the sense that the progress of each non-faulty process should not be prevented by other processes concurrently accessing the memory nor by failures incurred by other processes. The shared memory objects may suffer NR-Arbitrary faults [JCT98]. That is, a faulty object may fail to respond to an invocation, or may respond with an arbitrary value.

The strongest specification of a system's behavior under concurrent invocations is *linearizability*. The history of an object in a run $\alpha$ is *linearizable* if each operation invocation *op* in $\alpha$ can be reduced to a point between *op*'s invocation request and its corresponding response, such that the resulting sequence is *legal*. An *atomic* object is an object that has only linearizable histories.

It is possible to specify weaker behaviors under concurrent invocations. In this chapter, we consider *safe* registers. The history of a safe register is not necessarily linearizable, but if we remove all read operations that overlap write operations from the history of a safe register, the resulting history is linearizable.

## 6.3 The Safe Register Implementation

In this section we present a construction of a single-writer multi-reader safe register out of $n \geq 3t+1$ base registers, up to $t$ of which can experience NR-Arbitrary (Byzantine) faults.

For simplicity, we assume that the base registers are atomic (i.e., linearizable), although it is easy to see that the implementation also works correctly with regular registers (as defined in [Lam86]). To distinguish between the high-level reads/writes of the emulated register and the internal reads/writes of the base registers, we will denote the high-level operations of the emulated register using small capitals as READ and WRITE.

As dictated by the lower bound of Section 6.4.1, the WRITE operation invokes two rounds of write operations on the base registers. The first is called the *pre-write phase*, and the second, the *write* phase. Each value written to a base register in either phase is written together with a monotonically increasing timestamp, taken from a totally ordered set $TS$, with the minimum element $ts_0$. Thus, each base register $x_i$ holds a pair of pairs $x_i.pw$ and $x_i.w$, which hold the value-timestamp pairs written in the latest *pre-write* and *write* respectively. The shared base registers and their types are defined in Figure 6.1. In each phase, write operations are invoked on all base registers (for registers that did not yet respond in the first phase, the second phase operation remains pending until they do) and $n - t$ *ack*s are awaited. The WRITE implementation is shown in Figure 6.2.

```
Types: $TSVals = TS \times Vals$, with selectors $ts$, $val$;
       $X = TSVals \times TSVals$, with selectors $pw$, $w$;
Shared registers $x_i \in X$, $1 \leq i \leq n$,
       initialized to $\langle\langle ts_0, v_0\rangle, \langle ts_0, v_0\rangle\rangle$.
```

Figure 6.1: Base registers used in safe register construction.

```
WRITE($v$):
    choose $ts \in TS$ larger than previously used;
    || invoke write $\langle ts, v\rangle$ to $x_i.pw$, for each $x_i$;
    **wait** until $n - t$ base registers respond with $ack$;
    || invoke write $\langle ts, v\rangle$ to $x_i.w$, for each $x_i$;
    **wait** until $n - t$ base registers respond with $ack$;
    return $ack$;
```

Figure 6.2: The safe register WRITE emulation.

The READ implementation appears in Figure 6.3. The algorithm is *early-stopping*: it invokes at most $min(t + 1, f + 2)$ rounds of read operations on the base registers, which is a tight lower bound. The part of the code that ensures this bound on the number of rounds is shown in gray, and will be discussed later. We begin by discussing the correctness of the

```
Local variables:
    $P[1,\dots t+1], W[1,\dots,t+1] \subseteq \{1\dots n\} \times TSVals$, initially $\emptyset$
    $Sent[w] \subseteq \{1\dots n\}$ for $w \in TSVals$, initially $\emptyset$          /* Objects that have sent w */
    $C \subseteq TSVals$, initially $\emptyset$                                  /* Candidate values to return */
    $j \in$ Integers, initially 1

Predicate and macro definitions:
    $\mathsf{invalid}(\langle w\rangle) = \exists l \leq j : |\ \{\langle i', w'\rangle \in W[l] | w' \neq w\}\ | \geq 2t+1$
    $\mathsf{Valid}(S) = \{w \in S | \neg \mathsf{invalid}(w)\}$
    $\mathsf{highCand}(\langle ts, v\rangle) = \langle ts, v\rangle \in \mathsf{Valid}(C) \wedge (ts = sup\{ts' | \langle ts', v'\rangle \in \mathsf{Valid}(C)\})$
    $\mathsf{safe}(\langle ts, v\rangle) = |\ \{i|\ \exists l \leq j\ \exists \langle i, ts', v'\rangle \in P[l] \cup W[l] : ts' > ts \vee (ts' = ts \wedge v' = v)\}| \geq t+1$

READ():
(1)  while ($true$) do
(2)      || invoke read($x_i$) on all objects $x_i$;
             for each returned response $x_i$, store $\{\langle i, x_i.pw\rangle\}$ in $P[j]$ and $\{\langle i, x_i.w\rangle\}$ in $W[j]$;
(3)      wait until $|W[j]| \geq n - t \wedge$
             $\forall c \in Valid(C)$: $\mathsf{safe}(c) \vee (|\{k | \exists \langle k, w\rangle \in W[j]\} \setminus Sent[c]| \geq n - t)$;
         /* Values read in this round are candidates to return: */
(4)      $C \leftarrow \{w | \exists \langle k, w\rangle \in W[j]\}$
(5)      foreach $c \in Valid(C)$ do
(6)          $Sent[c] \leftarrow Sent[c] \cup \{k | \langle k, c\rangle \in W[j]\}$;
(7)          if ($\mathsf{highCand}(c) \wedge \mathsf{safe}(c)$) then
(8)              return $c.val$;
(9)      od
(10)     if ($\mathsf{Valid}(C) = \emptyset$) then
(11)         return some arbitrary value in $Vals$;
(12)     $j \leftarrow j + 1$;
(13) od
```

Figure 6.3: Early-stopping safe register READ emulation. Code for reducing the number of invoked rounds shown in gray.

algorithm, by looking only at the part of the code shown in black.

The reader repeatedly invokes rounds of read operations to the base objects. The algorithm stores the responses to the round $j$ read in the sets $P[j]$ and $W[j]$ as follows: when a response $x_i$ arrives for the round $j$ read, $\{\langle i, x_i.pw\rangle\}$ is stored in $P[j]$ and $\{\langle i, x_i.w\rangle\}$ in $W[j]$.

In each round, the algorithm waits for at least $n-t$ responses from base objects (line 3). The set $C$ includes values that are "candidates" for returning from the read. These are values that appear in the $x_i.w$ fields of responses gathered in the latest completed read round (line 4). Since each read round reads from at least $n-t \geq 2t+1$ base objects, any value that is not included in set $C$ was either not completely written before the READ began (its WRITE could have begun but couldn't have completed), or was already over-written (that is, a subsequent WRITE has begun). A value-timestamp pair from the set $C$ is called a *candidate*.

A candidate is deemed *invalid* if there is a round in which there are at least $2t+1$ object responses that do not include it. If a value is invalid, then it was either not completely written the READ began, or was already over-written. In either case, READ is not required to return it. The set $\mathsf{Valid}(C)$ includes all the candidates that are not invalid. Among those, the leading candidate to return is the one with the highest timestamp. This is because if several values were completely written before the READ begun, READ has to return the latest one, although the set $\mathsf{Valid}(C)$ may include some of the older ones because some of the objects responding to the READ may be out of date. (Recall that WRITE returns after having written to $n-t$ base objects, of which $t$ may be faulty, hence there may be $t$ correct objects that still have old values). The predicate $\mathsf{highCand}(c)$ is true for a candidate $c \in Valid(C)$ if $c$ has the highest timestamp in $\mathsf{Valid}(C)$.

Now, consider a candidate $c$ such that $\mathsf{highCand}(c)$ is true. It is *safe* for READ to return *c.val* if there are at least $t+1$ objects that have responded to any of the read rounds with either $c$ or a higher timestamped value. The predicate $\mathsf{safe}$ captures this condition. The following lemma shows that the values returned by READ operations comply with the safe register semantics:

**Lemma 22 (Safety).** *If* READ *does not overlap any* WRITE *operation, it returns the latest written value, or the initial value if no value was written.*

*Proof.* Let $\langle v, ts\rangle$ be the latest value whose WRITE completed before READ was invoked, or $\langle v_0, ts_0\rangle$ if no WRITE has completed. Assume further that no WRITE overlaps the READ. Therefore, there are at least $t+1$ correct objects that have $\langle v, ts\rangle$ in their $w$ and $pw$ fields throughout the duration of the READ operation. Since each round waits for at least $2t+1$ responses, at least one object responds with $\langle v, ts\rangle$ in each round, and $\langle v, ts\rangle$ is included in $C$ from the first time line 4 is executed onward. Moreover, $\langle v, ts\rangle$ never becomes invalid, so it is also in $\mathsf{Valid}(C)$. In particular, $\mathsf{Valid}(C)$ is never empty when checked in line 10, so the algorithm never returns in line 11. Finally, observe that no $\langle v', ts'\rangle \neq \langle v, ts\rangle$ can be $\mathsf{highCand}$ and $\mathsf{safe}$, because no correct object will return any value with $ts' > ts$ or $ts' = ts \wedge v' \neq v$. Hence, no other value can be returned in line 8. □

Since the only requirement imposed by the safe register semantics on register implementations is for every READ operation that does not overlap any WRITE operation to return the latest written value, or the initial value if no value was written, Lemma 22 implies that the algorithm in Figures 6.2 and 6.3 implements a safe register.

The part of the algorithm shown in black is also *live*, since it continues to gather responses in all the rounds and use these values to check the predicates. Eventually, all the correct base registers will respond to all the rounds. Thus, every candidate value will eventually be either invalidated (when $2t + 1$ objects will respond without it) or will become safe (when $t + 1$ objects will respond with it or with a later value). Nevertheless, this part of the code (without the gray part) can initiate an unbounded number of rounds. To avoid this, the gray part introduces an extra waiting condition that must be satisfied before the algorithm is allowed to initiate an additional round.

We now discuss the gray part of the code. The data structure $Sent[w]$ for a value-timestamp pair $w$ holds indices $i$ of base objects that have responded with $w$ in their $x_i.w$ field so far (it is updated in line 6). Consider a candidate $c$ that was sent by $k$ objects in previous rounds (that is, $|\mathsf{Sent}[c]| = k$); there are two cases:

1. *At least one of these $k$ objects is correct.* In this case, the *pre-write* phase of WRITE(c) must have completed before the current round is initiated (because the correct object responded with $c$ in its $x_i.w$ field in the previous round). Therefore, there are at least $t+1$ correct objects where $c$ was pre-written before the current read round. Each such object $i'$ will eventually respond with either $c$ or a higher timestamped value in its $x_{i'}.pw$ field. Therefore, waiting for $c$ to become safe does not violate liveness.

2. *These $k$ objects are all faulty.* In this case, it is safe to wait for responses in the current round from $n - t$ *other* objects. That is, it will not violate liveness to wait for the condition: $\{k | \exists \langle k, w\rangle \in W[j]\} \setminus Sent[c]| \geq n - t$ to hold true.

The additional waiting condition in line 3 waits for one of the above to hold, and therefore does not violate liveness, as stated in the following lemma:

**Lemma 23 (Non-blocking).** *The algorithm never blocks indefinitely the wait statement in line 3.*

*Sketch.* The first condition requires that $n - t$ responses arrive. Since there are at least $n - t$ correct objects, this condition will eventually be satisfied. The second conditions requires that for all $c \in Valid(C)$, at least one of the following become true: either $c$ will become safe, or $n - t$ objects that did not respond with $c$ in their $w$ fields in previous rounds will respond to the current round. If $\mathsf{Valid}(C)$ is empty, we are done. Otherwise, for every $c \in \mathsf{Valid}(C)$, one of these conditions will be satisfied as explained above. □

To understand why this waiting condition ensures that the algorithm invokes at most $t+1$ rounds, observe that once every candidate is either safe or invalid, READ returns. Consider a candidate. The first clause waits for it to become safe. If this is achieved, this candidate cannot stop the algorithm from returning. Otherwise, the algorithm waits on the second clause. After waiting for $n-t$ responses from objects that did not previously respond with $c$, either $\mathsf{Sent}[c]$ increases, or $c$ gets invalidated, because $2t+1$ objects respond without $c$. So for every candidate that is neither safe nor invalid at the end of round $j$, the set $\mathsf{Sent}[c]$ has grown $j$ times, and therefore includes at least $j$ elements. Once $\mathsf{Sent}[c]$ includes $t+1$ elements, $c$ is safe, and therefore the algorithm never reaches the end of round $t+1$ without returning.

The early-stopping property of the algorithm is more subtle, and is proven in the following Lemma.

**Lemma 24 (Early-Stopping).** *In every run in which $f$ objects exhibit Byzantine behavior, the* READ *algorithm invokes at most $min(t+1, f+2)$ rounds of read operations on base registers.*

*Sketch.* Consider the point just before line 12 is executed, that is, just before $j$ is being increased. The current value of $j$ is the number of read rounds already invoked. Since the algorithm did not return in line 11, $\mathsf{Valid}(C) \neq \emptyset$. Let $c$ be a highest timestamped candidate in $\mathsf{Valid}(C)$, i.e., $\mathsf{highCand}(c)$ holds. Since $c$ was not returned in line 8, $c$ is not safe. Since $c$ is not invalid, it was returned in every round $\leq j$, and each time by a new object (i.e., $|\mathsf{Sent}[c]|$ has increased $j$ times). Since $c$ is not safe, we know that $|\mathsf{Sent}[c]| < t+1$, and therefore $j < t+1$.

If $c$ is never returned by a correct object, then $\mathsf{Sent}[c]$ includes at most $f$ elements, and $j \leq f$. Otherwise, let $k < j$ be the first round during which $c$ is returned by a correct object (i.e., the first round during which a correct object is inserted into $\mathsf{Sent}[c]$). Then $c$ was sent by at least $k-1$ Byzantine faulty objects before round $k$, and there are at most $f-k+1$ Byzantine faulty objects that are not in $\mathsf{Sent}[c]$. Consider the set $S$ of objects that respond to rounds $k+1 \ldots j$. Since $\mathsf{Sent}[c]$ continues to increase in each round, $S$ includes at least $2t+j-k$ objects excluding the $k-1$ Byzantine objects that sent $c$ before round $k$. Since at most $f-k+1$ members of $S$ are Byzantine faulty, $S$ includes at least $2t+j-k-(f-k+1) = 2t+j-f-1$ correct objects.

Finally, since the pre-write phase of WRITE(c) has completed before round $k+1$ was initiated, there are at most $t$ correct objects that respond to rounds $k+1 \ldots j$ with values older than $c$ in their *pw* field. Therefore, if $S$ includes $2t+1$ correct objects, then $S$ includes at least $t+1$ that have either $c$ or a higher value in their *pw* field, and $c$ is safe. Since we assumed that $c$ is not safe, we get that $2t+j-f-1 < 2t+1$, that is, $j < f+2$.

We have shown that if READ does not return by the end of round $j$, then $j < t + 1$ and also $j < f + 2$. Therefore, READ invokes at most $f + 2$ rounds. □

From Lemmas 23 and 24, we get that READ is live, and always returns after $min(t + 1, f + 2)$ rounds in runs with $f$ Byzantine failures. Note that if there are additional benign (i.e., crash) failures, the algorithm is not slowed down. In invocations of READ that are not concurrent with any WRITE invocation, READ invokes at most $f + 1$ rounds. For space limitations, we do not prove this here, but it can be proven similarly to Lemma 24.

Moreover, we observe that in *synchronous* runs, the algorithm always terminates in two rounds. In order to formally make such a claim, we need to consider a partially synchronous (or timed-asynchronous) model [DLS88]. In such models, it is possible to wait for messages until a certain timeout. In periods when the system is synchronous, messages from correct processes always arrive by this timeout. Achieving good performance in synchronous runs of an asynchronous system is important, as such runs are common in practice.

## 6.4 Lower Bounds

In this section we prove lower bounds on the number of rounds that must be invoked in emulations of the weakest meaningful shared memory primitive, a wait-free binary *single-reader single-writer safe (1R1WS)* register. We define a *round of invocations* to be a collection of operations that are invoked *concurrently* on a number of base objects (at most once on each base object).

In Section 6.4.1 we show a lower bound on the number of rounds for emulating WRITE operations; we will consider only WRITE emulations that do not overlap any READ operation. Section 6.4.2, deals with READ emulations, both those overlapping WRITE operations and those that do not overlap any WRITE operation. For the sake of the READ lower bound, we consider algorithms in which the reader does not modify the base objects at all. This assumption is satisfied by our algorithm, as well as by previously suggested algorithms in our model [MAD02, ABO03, MR00, JCT98, Baz00]. In the Section 6.5.1, we informally discuss the implications of this assumption.

We consider a concurrent system $C$ implementing a wait-free 1R1WS register out of $n > 0$ base objects. We will assume that the object state cannot be modified concurrently by the reader and the writer, i.e., that the objects are read/write registers. This does not restrict the generality of the WRITE lower bounds since we will not require the 1R1WS emulation to terminate in the runs in which the writer and the reader take steps concurrently. It also does not affect the generality of the READ lower bound, since there we assume that only the writer is allowed to modify the object state. Consequently, since we are not seeking space lower bounds, we can assume a full information model in which all base objects have the same types and initial states, and are modified the same way in each round. (A concurrent system consisting of base objects $O_1, \ldots O_n$ of different types $T_1, \ldots, T_n$ can be emulated in this model by replacing each base object $o_i$ with a read/write register $O'_i$ that stores a tuple $\langle h^1, \ldots, h^n \rangle$, where each $h^i$ is a sequence of states $s_0^i s_1^i \ldots$ of the object $O_i$; all registers are initialized to the same initial value $s_0$ which is a tuple $\langle s_0^1, \ldots, s_0^n \rangle$ where $s_0^i$ is the initial state of $O_i$).

In order to prove lower bounds, it suffices to consider a subset of all possible runs. We will look at runs with exactly one READ operation and at most one WRITE operation, where the initial value of the emulated register is 0, and if WRITE occurs it writes the value 1. We denote the objects' initial value by $s_0$, and the last value written by the WRITE(1) operation as $s_1$.

### 6.4.1 Lower Bound on WRITE Emulations

In order to strengthen our results, we will replace the wait freedom requirement with a weaker *obstruction freedom* termination condition, which requires object operations that run by themselves long enough to eventually complete.

We will now prove that any algorithm implementing an obstruction-free 1R1WS register out of $n \leq 4t$ base objects up to $t$ of which can incur NR-Arbitrary failures will have runs with at least two invocation requests completed on the same object.

**Theorem 11.** *Suppose $n = 4t$. Then, for any $0 \leq f \leq t$, there exists a run of $C$ that includes a complete invocation of* WRITE(1) *and no other invocations, such that during* WRITE(1) *at least two invocation requests are completed on some correct base object.*

*Proof.* Let $S_1$, $S_2$, $S_3$, and $S_4$ be sets of base objects such that for each $i \geq 1$, $|S_i| = t$, and all four sets $S_1$, $S_2$, $S_3$, and $S_4$ are pairwise disjoint.

Assume by contradiction that in all runs consisting of a complete WRITE(1) invocation, and no other invocations, less than two invocations complete on each base object. Without loss of generality, assume that the first base objects to which WRITE(1) writes are those in $S_4$ (if it writes to less than $t$ objects, then it writes to a subset of $S_4$). Let $\alpha_1$ be a run of $C$ where all base objects in $S_1$ are initially crashed.

We construct a run $\alpha_2$ in which all the objects in $S_2$ are Byzantine faulty. Run $\alpha_2$ starts with all the activity of $\alpha_1$ except that the invocation requests targeted to the objects in $S_1$ do not occur in $\alpha_2$ until the WRITE(1) terminates (i.e., in $\alpha_2$ the threads that handle invocations on objects in $S_1$ are slowed down so that all the invocation requests issued in $\alpha_1$ are postponed). Since $\alpha_2$ is indistinguishable to the writer from $\alpha_1$, WRITE(1) also terminates in $\alpha_2$ after completing at most one invocation request at each base object $O_j$.

Next, we construct a run $\beta_1$ where all objects in a set $S_3$ are initially crashed. Run $\beta_1$ is derived from $\alpha_1$ by removing the following: (1) all base object invocation events except those occurring at the objects in the set $S_4$; and (2) the *ack* response returned by WRITE(1). Note that $\beta_1$ is a valid run of $C$ since it represents the situation in which the writer fails after receiving responses from some objects in the set $S_4$. We extend $\beta_1$ with a READ invocation and assume that the reader is correct. All the objects in $S_1$ and $S_2$ respond to read requests of the reader with $s_0$. The objects in $S_3$ are crashed, and hence do not respond. By obstruction-freedom, READ must eventually terminate and return some value. Let $\beta_2$ be an extension of $\beta_1$ ending with the reader's respond event. Since in $\beta_2$ the reader sees at most $t$ object in a state different from $s_0$, $\beta_2$ is indistinguishable to the reader from a run in which WRITE(1) has never been invoked and all objects returning $s_1$ are faulty. Therefore, READ must respond with 0 in $\beta_2$.

Finally, we construct a run $\alpha_3$ as follows: we extend $\alpha_2$ with the segment of $\beta_2$ that starts with the READ invocation request and ends with its corresponding response. Note that $\alpha_3$ is a valid run of $C$ because all the objects in $S_2$ are Byzantine faulty, and are therefore allowed to respond with $s_0$ even after WRITE(1) terminates, and since no *invoke* events occur at objects in $S_1$ in $\alpha_2$, the objects in $S_1$ are also allowed to respond with $s_0$. The responses of objects in $S_3$ are delayed until after the READ returns. By construction, $\alpha_3$ is indistinguishable to the reader from $\beta_2$, and therefore, the return value of READ must be 0 in $\alpha_3$. Since in $\alpha_3$ no reads are concurrent to writes, by safety, the history of $\alpha_3$ must be linearizable. However, this would require READ to return 1. A contradiction. □

### 6.4.2 Lower Bound on READ Emulations

In this section we show a lower bound on a number of rounds of base object invocations required to emulate READ operations of a binary 1R1WS register. We consider a system with $n = 3t + k$ base objects, $t$ of which can fail.

We will prove the following theorem:

**Theorem 12.** *For every algorithm $A$ emulating a 1R1WS register in a system with $n = 3t+k$ base objects, $t$ of which can suffer NR-Arbitrary failures, and in which the reader does not modify the base objects' states, there is a run of $A$ in which the* READ *emulation invokes $\lfloor t/k \rfloor + 1$ rounds of base object operations.*

Note: The special case where $k = 1$ represents an optimal resilience algorithm.

We begin by proving the following simple lemma:

**Lemma 25.** *For any $0 \leq f \leq t$, there is a finite run that includes a single complete* WRITE *operation, in which $f$ objects fail, and at the end of which $t$ correct base objects' state are $s_0$.*

*Proof.* Consider a run $\sigma$ in which $t$ (faulty) objects crash at the beginning of the run, and WRITE(1) is invoked. The WRITE must return without hearing from these $t$ objects. Let $\tau$ be the point in $\sigma$ at which WRITE returns. We construct a run $\sigma'$ that until point $\tau$ looks to the writer exactly like $\sigma$, but in which the requests sent to $t$ correct objects are delayed until after $\tau$. The remaining $n - t$ objects all abide by the protocol, and respond exactly as the $n - t$ correct objects do in $\sigma$. Since until $\tau$, $\sigma'$ is indistinguishable to the writer from $\sigma$, WRITE returns at $\tau$, before the delayed requests reach $t$ correct objects. Hence, when WRITE returns, these $t$ correct objects' states are still $s_0$. □

Without loss of generality, we will make the following assumption about the WRITE emulation: the last write operation invoked by WRITE to base objects attempts to write the same value to all base objects, and WRITE does not return before $2t + k$ of them respond. Modifying a given WRITE implementation in order to satisfy this assumption cannot violate safety, because it can only require writing more information to more objects; since we assume a full information protocol, no information is lost. Moreover, waiting for $2t + k$ responses does not violate liveness since at least $2t + k$ correct base objects are guaranteed to respond.

The above assumption implies that when a WRITE(1) operation completes, there exist $t + k$ correct base objects whose states are equal to the last value written by WRITE. We denote this state by $s_1$.

We now prove Theorem 12 by constructing a run in which the READ emulation invokes $\lfloor t/k \rfloor + 1$ rounds of base object operations. The run is constructed by induction, where we show that the algorithm is continuously forced to invoke more rounds. The proof is illustrated in Figure 6.4.

For $k > t$ the theorem trivially holds, because obviously at least one round is required in order to read a value from the register. We therefore assume that $k \leq t$. We construct, by induction on $i \leq t/k$, a run in which $i + 1$ rounds are required.

## Base case

A READ emulation begins by invoking read operations on all base objects, and waiting for responses. This is the first round. We construct three different runs which are indistinguishable to the reader after receiving $2t + k$ responses to the first round read request:

1. In run $\alpha_1$, a WRITE(1) operation completes before READ begins. Therefore, READ must return 1. When READ is invoked, at least $t + k$ correct objects' states are $s_1$. Without loss of generality, these are objects $o_1, \ldots, o_{t+k}$. Moreover, at least $t$ correct objects are in state $s_0$ (this is possible by Lemma 25). Without loss of generality, these are objects

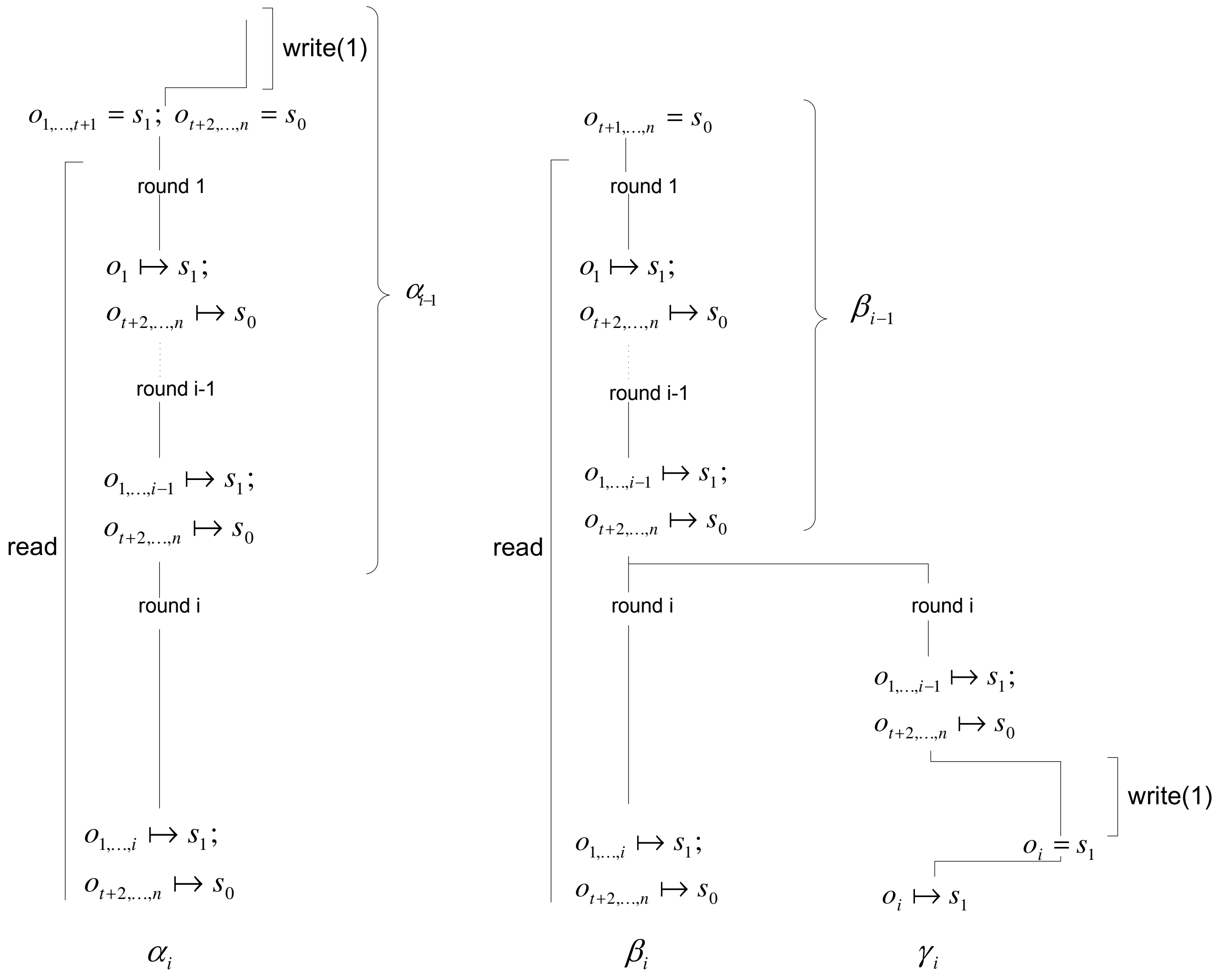
write(1)
$o_{1,\ldots,t+1} = s_1$; $o_{t+2,\ldots,n} = s_0$
read
round 1
$o_1 \mapsto s_1$;
$o_{t+2,\ldots,n} \mapsto s_0$
round i-1
$o_{1,\ldots,i-1} \mapsto s_1$;
$o_{t+2,\ldots,n} \mapsto s_0$
$\alpha_{i-1}$
round i
$o_{1,\ldots,i} \mapsto s_1$;
$o_{t+2,\ldots,n} \mapsto s_0$
$\alpha_i$
$o_{t+1,\ldots,n} = s_0$
read
round 1
$o_1 \mapsto s_1$;
$o_{t+2,\ldots,n} \mapsto s_0$
round i-1
$o_{1,\ldots,i-1} \mapsto s_1$;
$o_{t+2,\ldots,n} \mapsto s_0$
$\beta_{i-1}$
round i
round i
$o_{1,\ldots,i-1} \mapsto s_1$;
$o_{t+2,\ldots,n} \mapsto s_0$
write(1)
$o_i = s_1$
$o_i \mapsto s_1$
$o_{1,\ldots,i} \mapsto s_1$;
$o_{t+2,\ldots,n} \mapsto s_0$
$\beta_i$
$\gamma_i$

Figure 6.4: The runs constructed in the proof of Theorem 12 for $k = 1$. Notation: $o_j = s$ denotes the object $o_j$ being in the state $s$ at some point of an execution; $o_j \mapsto s$ denotes the object $o_j$ returning $s$ as a response to a *read* invocation.

$o_{t+k+1}, \ldots, o_{2t+k}$. In response to the first read round, $k$ correct objects, $o_1, \ldots, o_k$, return $s_1$ (if $k = 1$ then only $o_1$ returns $s_1$), $t$ correct objects, $o_{t+k+1}, \ldots, o_{2t+k}$ return $s_0$, and $t$ objects $o_{2t+k+1}, \ldots, o_n$ are faulty and return $s_0$. The remaining objects do not respond.

2. In run $\beta_1$, no WRITE ever occurs. Therefore, READ must return 0. When READ is invoked, all the correct objects are in state $s_0$. In this run, objects $o_1, \ldots, o_k$ are faulty and return $s_1$ in response to the first read round. $2t$ objects, $o_{t+k+1}, \ldots, o_n$, are correct and return $s_0$. The remaining objects do not respond (at least $k$ of them are correct).

3. In run $\gamma_1$, a WRITE(1) operation occurs concurrently with the READ operation. In this run, objects $o_1, \ldots, o_k$ are correct, and the first read request sent to them is delayed until after the WRITE operation completes and changes their states to $s_1$. They respond with $s_1$ to the first read round. Objects $o_{t+k+1}, \ldots, o_n$ are also correct, but they are still in state $s_0$ when the read request reaches them, because the read request reaches them before the first operation sent to them by the WRITE emulation. Therefore, they respond with $s_0$ to the first read round.

As a convention, the subscripts in the names of the runs denote the number of read rounds they include. Runs $\alpha_1$, $\beta_1$, and $\gamma_1$ all include one round of read operations.

Now, observe that in all three runs, the reader receives the same responses from base objects. In run $\alpha_1$, it is not allowed to return 0. In run $\beta_1$, it is not allowed to return 1. In run $\gamma_1$, it is not allowed to wait for more round 1 responses, because it already heard from $2t + k$ correct objects, and the remaining $t$ may be faulty. Therefore, a second round must be initiated. Moreover, note that in run $\gamma_1$, no objects are actually faulty (the reader must return because they may fail, but so far none have failed).

## Inductive step

Assume that $1 < i \leq t/k$. Our inductive hypothesis is as follows: There exist two finite runs $\alpha_{i-1}$ and $\beta_{i-1}$, both including $i - 1$ rounds of read, such that:

1. In $\alpha_{i-1}$ WRITE(1) has completed before the READ was invoked; objects $o_1, \ldots, o_{t+k}$ are correct and their state is $s_1$ from a time before the beginning of the READ onward; objects $o_{t+k+1}, \ldots, o_{2t+k}$ are correct and their state is $s_0$ throughout $\alpha_{i-1}$; and the remaining objects are faulty.

2. In $\beta_{i-1}$ no WRITE ever occurs; objects $o_1, \ldots, o_{(i-1)k}$ are faulty; and objects $o_{t+k+1}, \ldots, o_n$, are correct and their state is $s_0$ throughout $\beta_{i-1}$. For objects $o_{(i-1)k+1}, \ldots, o_{t+k}$ we do

not specify whether they are faulty or not since they do not participate in the run yet. At least $k$ of them must be correct, but we do not specify which ones.

3. The responses that the reader receives in both runs are as follows: In response to every round $j$ read, for $1 \leq j \leq i-1$, objects $o_1, \ldots, o_{jk}$ return $s_1$ and objects $o_{t+k+1}, \ldots, o_n$ return $s_0$. No other objects respond.

4. The reader invokes a round $i$ read immediately at the end of both of these runs.

It is easy to see that our construction of $\alpha_1$ and $\beta_1$ satisfies the induction hypothesis. We now show how to construct $\alpha_i$ and $\beta_i$ from $\alpha_{i-1}$ and $\beta_{i-1}$.

Note that by our assumption, $i \leq t/k$, hence $(i+1)k \leq t+k$, and at least $2k$ objects have not yet responded to any of the read rounds. The $i$th read round is invoked at the end of $\alpha_{i-1}$ and $\beta_{i-1}$. We construct three runs by adding $k$ responses to all the read rounds as follows:

1. Run $\alpha_i$ extends $\alpha_{i-1}$ by having the correct objects $o_{(i-1)k+1}, \ldots, o_{ik}$ respond to all the read rounds with their state $s_1$; and having all the objects that have responded in previous rounds respond the same way in round $i$.

2. Run $\beta_i$ extends $\beta_{i-1}$ by having the objects $o_{(i-1)k+1}, \ldots, o_{ik}$ become faulty and respond to all the read rounds with $s_1$; and having all the objects that have responded in previous rounds respond the same way in round $i$.

3. Run $\gamma_i$ extends $\beta_{i-1}$ with a WRITE(1) operation. The WRITE operation completes promptly, so that the final round of the WRITE occurs at objects $o_{(i-1)k+1}, \ldots, o_{ik}$ before these objects even receive the request for the round 1 read. Note that this is possible, since these objects have not responded to any request in $\beta_{i-1}$, and hence the request sent to them may have not reached them yet in $\beta_{i-1}$. These objects are all correct in $\gamma_i$, and they respond to all the read rounds with $s_1$. All the objects that have responded in previous rounds respond the same way in round $i$.

Again, we have a situation in which in all three runs, the reader receives the same responses from base objects. That is, these runs are indistinguishable to the reader. In run $\alpha_i$, the reader is not allowed to return 0. In run $\beta_i$, it is not allowed to return 1. In run $\gamma_i$, it hears from $2t+k$ correct objects in all the rounds, and is therefore not allowed to wait for more responses in any round, because the remaining objects may be faulty. Thus, round $i+1$ must be initiated. We add the initiation of round $i+1$ at the end of $\alpha_i$ and $\beta_i$, and they both satisfy the induction hypothesis. Theorem 12 follows.

Note that in run $\gamma_i$, $(i-1)k$ objects are faulty. Therefore, we have also proven the following lower bound for early-stopping algorithms:

**Theorem 13.** *Consider an algorithm $A$ emulating a 1R1WS register in a system with $n = 3t + k$ base objects, $t$ of which can suffer NR-Arbitrary failures, and in which the reader does not modify the base objects' states. For every $1 \leq i \leq \lfloor t/k \rfloor$, there is a run of $A$ in which $(i-1)k$ objects fail and the* READ *emulation invokes $i + 1$ rounds of base object operations.*

Note also that in run $\beta_i$, $ik$ objects are faulty and no READ operation overlaps any WRITE operation. Therefore we get the following lower bound for invocations of READ that do not overlap any WRITE:

**Theorem 14.** *Consider an algorithm $A$ emulating a 1R1WS register in a system with $n = 3t + k$ base objects, $t$ of which can suffer NR-Arbitrary failures, and in which the reader does not modify the base objects' states. For every $1 \leq i \leq \lfloor t/k \rfloor$, there is a run of $A$ in which $ik$ objects fail and the* READ *emulation invokes $i + 1$ rounds of base object operations.*

When $k = 1$, we get that for $0 \leq f \leq t$, the lower bound on the number of rounds required to emulate READ in runs with $f$ failures is $min(t + 1, f + 2)$. If a READ invocation does not overlap any WRITE operation, then the lower bound is $f + 1$. Our algorithm shows that these bounds are tight.

## 6.5 Conclusions

We have studied asynchronous implementations of wait-free shared memory objects from base objects that can suffer NR-Arbitrary faults. This failure model is important in capturing much recent work on scalable widely-distributed systems that are based on either "light" replicated servers (e.g., Fleet [MR00], Agile Store [LAV01], and Coca [ZSR02]) or the emerging technology of Storage Area Networks.

In this model, we have presented an optimal resilience wait-free construction for the first time: our construction uses $3t + 1$ base objects, $t$ of which can fail. We have also shown that optimal resilience algorithms have an inherent cost: we proved a lower bound of two rounds for emulating WRITE operations with resilience of $t \geq n/4$. This is in contrast to algorithms tolerating $t < n/4$ NR-Arbitrary faults, which can emulate WRITE operations in a single round. Moreover, we have shown a lower bound of $min(t + 1, f + 2)$ rounds for emulating READ operations in runs with $f$ failures in systems where the reader does not modify the base objects. Whether this lower bound still holds when readers are allowed to modify the base objects remains an open problem. However, we conjecture that even if readers can modify the base objects, it still holds that either the READ or the WRITE emulation must take $min(t + 1, f + 2)$ rounds. We further discuss this conjecture in Section 6.5.1.

Both of the bounds we have proven are *tight*: our shared register construction achieves both. Moreover, in runs that are synchronous or in which all failures are benign, our algorithm emulates WRITE operations in at most two rounds.

### 6.5.1 Allowing Readers to Modify Objects

Our lower bound on the number of rounds for READ assumes that the reader does not modify the base objects. We now revisit this assumption. Consider an algorithm in which the reader modifies the base objects and the writer reads information from them. How can such an algorithm be more efficient than an algorithm in which the reader is not allowed to modify the base objects? Conceivably, the reader may be able to signal to the writer that a read is in progress, and the writer could conceivably use this signal in order to refrain from writing to base objects while the reader is reading them. Observe that indeed, our lower bound proof made use of the fact that WRITE can occur concurrently with the READ.

Whether expediting the READ emulation by allowing readers to write and writers to read is possible or not remains an open problem. However, we believe that in order to allow some form of meaningful communication from the reader to the writer, one would need the abstraction of a 1R1WS register, where the READ emulation is the writer and the WRITE emulation is the reader. Intuitively, a safe register is needed in order for the reader to be able to signal to the writer that read is in progress, and for the writer to be able to distinguish the case that the reader never signaled that read is in progress from the case that the reader did signal so before the WRITE began. We conjecture that no form of communication weaker than a safe register can help reduce the cost of a READ emulation. Therefore, we believe that a safe register in one direction must be emulated at the "full cost" before a safe register in the other direction can be emulated faster. We therefore conjecture that if it is possible to expedite the read in this manner, then the WRITE emulation needs to invoke at least $\lfloor t/k \rfloor + 1$ rounds of read operations on base objects. Formally, we make the following conjecture:

**Conjecture 1.** *For every algorithm $A$ emulating a 1R1WS register in a system with $n = 3t + k$ base objects, $t$ of which can suffer NR-Arbitrary failures, there is a run of $A$ in which either the* READ *emulation or the* WRITE *emulation invokes $\lfloor t/k \rfloor + 1$ operation rounds.*

# Chapter 7

# Deconstruction Revisited

In this chapter we present an implementation of a Paxos-like agreement protocol in the shared memory environment with objects prone to NR-arbitrary failures. Our implementation follows the deconstruction approach of Chapter 4: We start by specifying two building blocks: (1) a weaker variant of the ranked register object, called a *safe ranked register* (Section 7.1), and (2) a write-once atomic register (Section 7.2). We then show an agreement implementation based on these two building blocks (Section 7.3), and finally, their construction tolerating up to $n/5$ NR-arbitrary object failures (Sections 7.4 and 7.5.

## 7.1 The safe ranked register object

In this section we present a weaker variant of the ranked register object implementable in the shared memory model with NR-arbitrary memory object failures. Naturally, in such an environment it might be impossible to reliably recover a register's content if it is being simultaneously accessed by several faulty processes. Also, in some situations it can be impossible to discern the rank with which a certain value has been written, albeit the value itself can be reliably chosen (e.g., a reader might see $t+1$ pairs $\langle r_i, v_i \rangle$ such that all $v_i$'s are the same whereas at least two $r_i$'s are different).

To address these possibilities, we introduce a weaker type of the ranked register object, called a *safe* ranked register. Informally, the read from a safe ranked register object is guaranteed to return a meaningful value only if the previously committed value has not been (or attempted to be) overwritten by a different value. The formal specification of the safe ranked register object is given below:

Let $Vals = Vals$. A safe ranked register object is a multi-reader, multi-writer shared register with two operations: $\mathsf{rr}\text{-}read(r)_i$ by process $i$, $r \in Ranks$, whose corresponding reply is $value(v)$, where $v \in Vals \cup \{\perp\}$. And $\mathsf{rr}\text{-}write(V)_i$ by process $i$, $V \in RVals$, whose reply is either $commit_i$ or $abort_i$. The safe ranked register object is required to satisfy the following properties:

**Property 10 (Integrity).** *If a* rr*-read operation $R$ returns $v \neq \bot$, then there exists a* rr*-write operation $W =$* rr*-write$(\langle r, v\rangle)$.*

**Property 11 (Safety).** *Let $W =$* rr*-write$(\langle r_1, v_1\rangle)_i$ be a* rr*-write operation that commits, and let*
*$R =$* rr*-read$(r_2)$ be a* rr*-read operation such that $r_2 > r_1$. If there does not exist a* rr*-write operation* rr*-write$(\langle r, v\rangle)$ such that $r > r_1$ and $v \neq v_1$, then $R$ returns the value $v_1$.*

**Property 12 (Non-Triviality).** *If a* rr*-write operation $W$ invoked with the rank $r_1$ aborts, then there exists a* rr*-read(*rr*-write) operation with rank $r_2 > r_1$ which is invoked before $W$ returns.*

**Property 13 (Liveness).** *If an operation (*rr*-read or* rr*-write) is invoked, then it eventually returns.*

As with the regular ranked register object we assume that each run satisfies rank uniqueness (Definition 1). Note that the safe ranked register object properties imply that given a rr-*write* operation $W =$ rr-*write*$(\langle r, v\rangle)$ that commits, a rr-*read* operation $R$ with rank $r > r$ may return $\bot$ only if it is concurrent to a rr-*write* operation $W' =$ rr-*write*$(\langle r', v'\rangle)$ such that $r' > r$ and $v' \neq v$.

## 7.2 A write-once atomic register

The ranked register object based Consensus implementation for NR-crash (see Figure 4.1) uses an additional read/write register to store the decision value. This register enables non-leader processes to obtain the decision value without intervening with the leader's DECIDE procedure. It is therefore, essential to ensure the termination property of Consensus. However, as we have seen in Chapter 6 implementing a register out of objects prone to NR-arbitrary failures is costly. Fortunately, once the Consensus value is fixed, it will never change, and therefore, it is enough to provide a read/write register with a weaker, write-once semantics. In this section we give a specification of this register. Its fault-tolerant construction is shown in Section 7.5.

A multi-reader/multi-writer read/write register is called a *write-once atomic register* if each its admissible execution $\sigma$ satisfies the following

**Property 14.** *If $\sigma'$ is the maximal admissible prefix of $\sigma$ in which no more than one value is written, then $\sigma'$ is an atomic register execution.*

## 7.3 Consensus using a safe ranked register object

The agreement implementation using a safe ranked register object is shown in Figure 7.1. It employs a method that is similar to that in Figure 4.1.

```
Shared: Safe ranked register object $rr$, initialized by $\mathsf{rr}\text{-}write(\langle r_0, v_0\rangle)$ which commits;
Write-once atomic register $decision$, with values in $Vals$, initialized to $\perp$
Local: $V \in Vals \cup \{abort\}$, $r \in Ranks$;

Process $i$:

propose($inp$), $Vals \to Vals$
    $r \leftarrow r_0$;
    **while**($true$) do
        $v \leftarrow decision.read$;
        **if** ($v \neq \perp$) **then**
            return $v$;
        **if** ($\mathcal{L}_i$.isLeader()) **then**
            $r \leftarrow$ chooseRank($r$);
            $V \leftarrow$ DECIDE($\langle r, inp\rangle$);
            **if** ($V \neq abort$) **then**
                $decision.write(V)$;
                return $V$;
        **fi**
    **od**

Function DECIDE($\langle r, v\rangle$), $RVals \to (Vals \cup \{abort\})$:
    $V \leftarrow rr.\mathsf{rr}\text{-}read(r)_i$;
    if ($V = \perp \vee V = v_0$) then
        $V \leftarrow v$;
    if ($rr.\mathsf{rr}\text{-}write(\langle r, V\rangle)_i = commit$) then
        return $V$;
    fi
    return $abort$;
```

Figure 7.1: Consensus using a safe ranked register object, a write-once register and $\mathcal{L}$

**Lemma 26.** *Let $\alpha$ be an execution of the Consensus construction in Figure 7.1. Let $W_1 = rr.\mathsf{rr}\text{-}write(\langle r_1, v_1\rangle)$, be the lowest ranked $\mathsf{rr}$-write invocation that commits in $\alpha$. Then, if a $\mathsf{rr}$-write invocation $W = rr.\mathsf{rr}\text{-}write(\langle r, v\rangle)$, such that $r > r_1$, is invoked in $\alpha$, then $v = v_1$.*

*Proof.* Let $\mathcal{W}$ be the set of $\mathsf{rr}$-*write* operations invoked with a rank higher than $r_1$ in $\alpha$. We show by induction on the size of $\mathcal{W}$ that each $W \in \mathcal{W}$ writes the value it reads from $W_1$.

Base case: Let $\mathcal{W} = \{W\}$ where $W = \mathsf{rr}\text{-}write(\langle r, v\rangle)$. By the code, $W$ is preceded by a $\mathsf{rr}$-*read* operation $R = \mathsf{rr}\text{-}read(r)$ that returns $v$. Since $r > r_1$, and no other $\mathsf{rr}$-*write* operations with a rank higher than $r_1$ were invoked, by safety of $rr$, $R$ returns $v_1$. Hence, $v = v_1$ as needed.

Inductive step: Assume that the result holds for any set $\mathcal{W}$ such that $|\mathcal{W}| \geq 1$, and consider $\mathcal{W}' = \mathcal{W} \cup \{W'\}$. By the code, $W'$ is preceded by a $\mathsf{rr}$-*read* operation $R' = \mathsf{rr}\text{-}read(r')$. By the induction hypothesis, no value $v \neq v_1$ was ever written with rank $> r_1$, and by rank uniqueness no value but $v_1$ can be written with rank equal to $r_1$. Therefore, by safety of $rr$, $R'$ returns $v_1$ as needed. □

**Corollary 1.** *Let $\alpha$ be an execution of the Consensus construction in Figure 7.1. Let $W_1 = rr.\mathsf{rr}\text{-}write(\langle r_1, v_1\rangle)$, be the lowest ranked $\mathsf{rr}$-write invocation that commits in $\alpha$. Then any write to the decision register writes $v_1$.*

*Proof.* By the code, a value $v$ can be written to the *decision* register only after a $\mathsf{rr}$-*write* operation $W = \mathsf{rr}\text{-}write(\langle r, v\rangle)$ where $r > r_1$ commits. By Lemma 26, $v = v_1$. □

The agreement property of Consensus follows immediately from the above results:

**Lemma 27 (Agreement).** *The construction in Figure 7.1 satisfies Agreement.*

Next, we show validity:

**Lemma 28 (Validity).** *The construction in Figure 7.1 satisfies Validity.*

*Proof.* Let $\alpha$ be an execution of the construction in Figure 7.1. By initialization, $\alpha$ starts with the RR-WRITE operation $W_0$ that writes $\langle r_0, v_0\rangle$. Let $\mathcal{W}$ be the set of $\mathsf{rr}$-*write* operations invoked in $\alpha$. We show by induction on the size of $\mathcal{W}$ that each $W \in \mathcal{W}$ writes the initial value of some process.

Base case: Let $\mathcal{W} = \{W\}$ where $W = \mathsf{rr}\text{-}write(\langle r, v\rangle)_i$. By the code, $W$ is preceded by a $\mathsf{rr}$-*read* operation $R = \mathsf{rr}\text{-}read(r)$. Since $r > r_0$, and no other $\mathsf{rr}$-*write* operations with a rank higher than $r_0$ have yet been invoked, by safety of $rr$, $R$ returns $v_0$. By the code, $v = inp_i$ as needed.

Inductive step: Assume that the result holds for any set $\mathcal{W}$ such that $|\mathcal{W}| \geq 1$, and consider $\mathcal{W}' = \mathcal{W} \cup \{W'\}$ such that $W' = \mathsf{rr}\text{-}write(\langle r', v'\rangle)_i$. Again, by the code, the value $v'$ written by $W'$ is determined by the value returned by the preceding $\mathsf{rr}$-*read* operation $R' = \mathsf{rr}\text{-}read(r')$. By integrity, $R$ returns one of the following: (1) $\perp$, (2) $v_0$, or (3) the value $v$ written by some $W \in \mathcal{W}$. If (1) or (2) is true, then $v' = inp_i$ as needed. Otherwise, by the induction hypothesis, it must be the input value of some process $j$, so that $v' = v$ as required. □

Finally, we show termination:

**Lemma 29 (Termination).** *The construction in Figure 7.1 satisfies Termination.*

*Proof.* At the latest, when some correct process $\ell$ becomes a unique perpetual leader, it will go through the successive iterations of the loop in the propose code, increasing its rank at each iteration. Eventually, its rank will become the highest among all the ranks passed to the operations invoked so far. By non-triviality, as soon as this happens, $\ell$ will succeed to write a value $v \neq \bot$ to $rr$ and to *decision*. Each other process, that has not yet decided, will eventually invoke *decision.read* that does not overlap with $decision.write(v)_\ell$, and therefore, by Property 14 will read the value $v \neq \bot$ and terminate. □

## 7.4 Implementing a safe ranked register object

In this section we present a wait-free implementation of a safe ranked register from regular ranked register objects that may incur NR-arbitrary failures. The register supports an unbounded number of clients. The construction pseudocode appears in Figure 7.2. The construction assumes $n > 5t$ shared ranked register objects up to $t$ of which can incur NR-arbitrary failures.

We now argue the implementation correct. We first show safety (Property 11).

**Lemma 30 (Safety).** *The implementation in Figure 7.2 satisfies safety (Property 11).*

*Proof.* Let $W_1 = \text{RR-WRITE}(\langle r_1, v_1 \rangle)$ be a rr-*write* operation that commits, and let $\text{RR-READ}(r_2)$ such that $r_2 > r_1$. Assume that there does not exist a rr-*write* operation $W = \text{rr-}write(\langle r, v \rangle)$ such that $r > r_1$ and $v \neq v_1$.

By the rr-*write* code, the $rr_i$.rr-$write(\langle r_1, v_1 \rangle)$ invocation on at least $n - 2t$ base registers $rr_i$, of which at least $n - 3t > 2t$ are correct, returns *commit*. Consequently, by safety of $rr_i$, $rr_i$.rr-$read(r_2)$ must see[1] $rr_i$.rr-$write(\langle r_1, v_1 \rangle)$ for at least $t + 1$ base registers $rr_i$.

Since by assumption, $v_1$ is never overwritten with a different value by a rr-*write* operation with rank $> r_1$, and by rank uniqueness, the only rr-*write* operation with rank $= r_1$ is $W_1$, there must be at least $t + 1$ base registers $rr_i$ such that $rr_i$.rr-$read(r_2)$ returns $\langle r, v_1 \rangle$ where $r \geq r_1$.

We show that that there does not exist a pair $\langle s, u \rangle$ where $s \geq r_1$ and $u \neq v_1$, such that the test in line 5 of the RR-READ implementation yields true for $\langle s, u \rangle$. Assume to the contrary that such $\langle s, u \rangle$ exists. Since $\langle s, u \rangle$ satisfies the test in line 5, there exist at least $t + 1$ base registers $rr_j$ that responded with $\langle s', u \rangle$ such that $s' \geq s$ to $rr_j$.rr-$read(r_2)$. Since at least one of the $rr_j$ registers is correct, by safety of $rr_j$, there exists a rr-*write* invocation $W''$ with rank $s'' \geq s'$ that writes the value $u \neq v_1$. However, we know that $s'' \geq s' \geq s \geq r_1$, and since $u \neq v_1$, by rank uniqueness, $s'' > r_1$, thus contradicting the assumption that no such $W''$ may ever exist.

[1] In the sense of Definition 2 in Section 4.2.

```
Shared: ranked register objects $rr_j$, $1 \leq j \leq n$

Local to the reader: $S \subseteq \{1 \ldots n\} \times \mathit{RVals}$, initially empty.
RR-READ($r$):
(1) $S \leftarrow \emptyset$;
(2) $\|$ invoke rr-$\mathit{read}(r)$ on $rr_i$ for each $i$;
(3)     for each response $V_i$ returned by $rr_i$, store $\langle i, V_i \rangle$ in $S$;
(4) wait until $|S| \geq n - t$;
(5) if $(\exists V \in \mathit{Vals}\ \exists R \in \mathit{Ranks}\ [R = \mathit{max}\{r' : |\{k : \langle k, r'', V \rangle \in S \wedge r'' \geq r'\}| \geq t+1\}])$ then
(6)     return any value $V$ satisfying the above test;
(7) return $\perp$;

Local to the writer: $S \subseteq \{1 \ldots n\} \times \{\mathit{commit}, \mathit{abort}\}$, initially empty.
RR-WRITE($\langle r, v \rangle)_i$:
(1) $S \leftarrow \emptyset$;
(2) $\|$ invoke rr-$\mathit{write}(\langle r, v \rangle)$ on $rr_i$ for each $i$;
(3)     for each response $\mathit{resp}_i$ returned by $rr_i$, store $\langle i, \mathit{resp}_i \rangle$ in $S$;
(4) wait until $|S| \geq n - t$;
(5) if $(|\{k : \langle k, \mathit{abort} \rangle \in S\}| \geq t + 1)$ then
(6)     return $\mathit{abort}$;
(7) return $\mathit{commit}$;
```

Figure 7.2: A wait-free construction of a safe ranked register object out of $n \geq 5t+1$ regular ranked register objects

Finally, since the test in line 5 is satisfied for $\langle r_1, v_1 \rangle$, and for any pair $\langle s, u \rangle$ satisfying the test in line 5 must hold $s \geq r_1$ and $u = v_1$, the return value of RR-READ($r_2$) must be $v_1$ as needed. □

Next, we show the remaining properties:

**Lemma 31 (Integrity).** *The implementation in Figure 7.2 satisfies integrity (Property 10).*

*Proof.* By the RR-READ code, if $V \neq \perp$ is returned, then there exist at least $t + 1$ base ranked register objects that returned $\langle *, V \rangle$ one of which, say $rr_j$, is correct. By safety of $rr_j$, there exists $rr_j$.rr-$\mathit{write}(\langle r, V \rangle)$. By the code, $rr_j$.rr-$\mathit{write}(\langle r, V \rangle)$ can only be invoked by RR-WRITE($\langle r, V \rangle$) as needed. □

**Lemma 32 (Non-Triviality).** *The implementation in Figure 7.2 satisfies non-triviality (Property 12).*

*Proof.* By the RR-WRITE implementation, a RR-WRITE operation $W = \text{RR-WRITE}(\langle r, v\rangle)$ returns *abort* only if rr-*write*$(\langle r, v\rangle)$ operations applied to at least $t+1$ base ranked register object have returned *abort*. Since at least one of these registers, say $rr_j$, is correct, by non-triviality of $rr_j$, there exists a rr-*read* (rr-*write*) invocation $rr_j$.rr-$read(r')$ ($rr_j$.rr-$write(\langle r', *\rangle)$) such that $r' > r$. By the code, $rr_j$.rr-$read(r')$ ($rr_j$.rr-$write(\langle r', *\rangle)$) can only be invoked by RR-READ$(r')$ (RR-WRITE$(\langle r', *\rangle)$) as needed. □

**Lemma 33 (Liveness).** *The implementation in Figure 7.2 satisfies liveness (Property 13).*

*Proof.* Obviously, both the RR-READ and RR-WRITE implementations can never get stuck unless a rr-*read* or rr-*write* from some base register gets stuck. However, this cannot happen since the base registers satisfy liveness. □

We proved the following

**Theorem 15.** *The implementation in Figure 7.2 is an implementation of the safe ranked register object out of $n > 5t$ regular ranked register objects upto $t$ of which can incur NR-arbitrary failure.*

## 7.5 Implementing a write-once atomic register

A fault-tolerant construction of a write-once atomic register appears in Figure 7.3. It utilizes $n > 4t$ write-once atomic registers upto $t$ of which can incur NR-arbitrary failures.

We now prove the construction correct. Let $\sigma$ be an admissible execution of the construction in Figure 7.3, and $\sigma'$ be the maximal admissible prefix of $\sigma$ such that there exists a value $v \in Vals$ satisfying the following: If *write*$(u)$ in invoked in $\sigma'$, then $u = v$.

**Lemma 34.** *Each* READ *operation $R$ in $\sigma'$ returns either $v$ or $\perp$. Moreover, if $R$ returns $v$, then there exists a* WRITE *operation $W =$* WRITE$(v)$ *such that $W$ is invoked before $R$ returns; and if $R$ returns $\perp$, then for no* WRITE *operation $W$, $W$ returns before $R$ is invoked.*

*Proof.* First, for no value $v' \neq v$, $v'$ is ever returned by READ as for otherwise, there exist $t+1$ base registers that returned $v'$, and therefore, some correct base register $x_j$ returned $v'$. By atomicity of $x_j$, there exists $w' = write(v')$ invoked on $x_j$ such that $w'$ is invoked before the read from $x_j$ returns. In turn, $w'$ must be invoked by $W' = \text{WRITE}(v')$ such that $W'$ is

```
Shared: Write-once atomic registers $x_i \in Vals$, $1 \leq i \leq n$, initialized to $\bot$.
Local: $Resp \subseteq \{1 \ldots n\} \times Vals$, initially empty;

WRITE($v$):
    || invoke $write(v)$ on $x_i$ for each $i$;
    wait until $n - t$ base registers respond with $ack$;
    return $ack$;

READ():
    || invoke $read()$ on $x_i$ for each $i$;
        for each response $resp_i$ returned by $x_i$, store $\langle i, resp_i \rangle$ in $Resp$;
    wait until $|Resp| \geq n - t$;
    if $\exists v \in Resp$ such that $|\{k : \langle k, v \rangle \in Resp\}| \geq t + 1 \land v \neq \bot$ then
        return $v$;
    return $\bot$;
```

Figure 7.3: A wait-free construction of the write-once atomic register out of $n > 4t + 1$ write-once atomic registers

invoked before $R$ returns contradicting the assumption that no values other than $v$ are ever written in $\sigma'$.

In the same vein, if $R$ returns $v$, then there exist $t + 1$ base registers that returned $v$, and therefore, some correct base register $x_j$ returned $v$. By atomicity of $x_j$, there exists $w = write(v)$ invoked on $x_j$ such that $w$ is invoked before the read from $x_j$ returns. In turn, $w$ must be invoked by $W =$ WRITE($v$) such that $W$ is invoked before $R$ returns.

Finally, suppose that $R$ returns $\bot$. Assume to the contrary, that there exists $W =$ WRITE($v$) that returns before $R$ is invoked. At the time $W$ returns, *write*($v$) was complete on at least $n - t \geq 3t + 1$ base registers $x_j$ of which at least $2t + 1$ are correct. Since no other value $v' \neq v$ is ever written to each $x_j$ in $\sigma'$, and because $R$ invokes *read* on at least $n - t$ base objects, at least $t + 1$ correct base objects will respond with $v$ to the *read* operations invoked by $R$. Of the remaining $n - 2t$ objects, at most $t$ may respond with arbitrary values, and the last $n - 3t$ will respond with $v$ or $\bot$. Thus, the only value that can appear $t + 1$ times in *Resp* in addition to $v$ is $\bot$. By the algorithm, in this case, the reader will always choose to return $v$ in contradiction to the assumption that $R$ returns $\bot$. □

**Lemma 35.** *$\sigma'$ satisfies atomicity.*

*Proof.* Our proof strategy is to construct a permutation $\pi$ of the operations invoked in $\sigma'$

that is both consistent with the $\rightarrow$ relation, and legal with respect to the read/write semantics. $\pi$ is constructed as follows: First, let $\pi_1$ be a permutation of the operations invoked in $\sigma'$ that is consistent with the $\rightarrow$ relation. Next, based on the fact that by Lemma 34, each READ operation in $\sigma'$, and therefore in $\pi_1$, returns either $\perp$ or $v$, we transform $\pi_1$ into $\pi_2$ using the procedure outlined below:

```
Do
  $\pi_2 \leftarrow \pi_1$;

  (1) For each READ operation $R$ in $\sigma'$ such that $R$ returns $\perp$, and there
  exists a WRITE operation $W$ that precedes $R$ in $\pi_1$, remove $R$ from its
  original location and insert it before $W$.

  (2) For each READ operation $R$ in $\sigma'$ such that $R$ returns $v$, and there
  exists a WRITE operation $W$ that follows $R$ in $\pi_1$, remove $R$ from its
  original location and insert it after $W$.
Until $\pi_2 = \pi_1$.
```

We show that $\pi = \pi_2$ is the permutation required by atomicity. First, we show that $\pi$ is consistent with the $\rightarrow$ relation. Indeed, by Lemma 34, each READ operation that returns $\perp$ must be invoked before any WRITE operation returns, the first transformation above leaves $\pi_1$ consistent with $\rightarrow$. Also, by Lemma 34, each READ operation that returns $v$ must return after any WRITE is invoked. Therefore, the second transformation above keeps $\pi_1$ consistent with $\rightarrow$ as well. Finally, since the transformation procedure does not affect the relative order of writes, the $\pi$ is consistent with the $\rightarrow$ relation.

Last, we show that $\pi$ is legal. Indeed, the transformation procedure implies that $\pi = \pi'\text{WRITE}(v)\pi''$ where $\pi'$ consists of only READ operations that return $\perp$, and $\pi''$ consists of WRITE operations that write $v$ and READ operations that return $v$. Hence, $\pi$ is legal. $\Box$

Since both WRITE and READ are obviously non-blocking, the construction in Figure 7.3 is wait-free. Thus, we proved the following

**Theorem 16.** *The construction in Figure 7.3 is a wait-free construction of a write-once atomic register out of $n > 4t$ write-once atomic registers upto $t$ of which can incur NR-arbitrary failure.*

# Part III

# Realizing the Data-Centric Fault-Tolerance

# Chapter 8

# Aquarius: A Storage-Centric approach to CORBA Fault-Tolerance[1]

## 8.1 Introduction

The Internet provides abundant opportunity to share resources, and form commerce and business relationships. Key to sharing information and performing collaborative tasks are tools that meet client demands for reliability, high availability, and responsiveness. Many techniques for high availability and for load balancing were developed aiming at small to medium clusters. These leave much to be desired when facing highly decentralized settings.

In order to take the existing, successful approaches a step forward towards large scale distributed systems, we identify two core challenge areas related to information technology tools.

The first is attention to scale and dynamism, in order to exploit reliability and survivability techniques in the networks of today and the future. This dissertation manifests the realization that replication based on techniques borrowed from the group communication world fail to scale beyond a few dozens of servers, and incur a serious cost of cross-server monitoring for failures. In this chapter we establish the storage-centric paradigm as a viable alternative to the group communication approach by providing its concrete and detailed implementation and performance assessment.

The second is deployment in real settings and providing an evolution path for legacy software. While we strive to keep the work general, we focus on CORBA [Obj99] as a development platform. This choice is made so as to provide for inter-operability and uniformity, and comply with state-of-the-art heterogeneous middlewares. CORBA is a leading standard

[1]This chapter is based on the WRSM'03 paper by Chockler, Malkhi, Merimovich and Rabinowitz [CMMR03].

for bridging object oriented and distributed systems technologies, simplifying development of distributed applications. Our results provide important insights that may impact the emerging Fault-Tolerant CORBA (FT-CORBA) standard [Obj00].

In this chapter we introduce *Aquarius*, a fault tolerant CORBA software that answers the two challenges mentioned above. First, Aquarius employs replication techniques from the previous chapters for survivability and scalability. Second, the design seamlessly wraps legacy software to provide a smooth migration path for robustifying existing services. The architecture is demonstrated using a test-case replicated SQL database.

### 8.1.1 Designing Robust Services using Storage-Centric Replication

Consider a typical service program that is accessible by many clients over the network. The challenge is to robustify the service for high availability and load balancing with little or no intervention to existing client or server code.

The *storage-centric* approach regards the service as a shared object which is manipulated by multiple clients. Copies of the object reside on a collection of persistent storage servers, accessed by an unbounded universe of transient client processes. In order to coordinate updates to different copies of an object, clients perform an agreement protocol similar to that described in Chapters 4 (Chapter 7 in case of NR-arbitrary failures).

The design puts minimal additional functionality on data servers, who neither communicate with one another, nor are aware of each other. Essentially, each server needs a thin wrap that provides a facility for storing and retrieving temporary 'meta-data' per object. Clients are also not heavy. Their interaction is through rounds of remote invocations on quorums of servers. The approach offers great simplicity for constructing fault-tolerant distributed systems featuring a high degree of decentralization and scale.

More concretely, this approach has several important advantages. First, it alleviates the cost of monitoring replicas and reconfiguration upon failures. Second, it provides complete flexibility and autonomy in choosing for each replicated object its replication group, failure threshold, quorum system, and so on. In contrast, most implementations of state machine replication pose a central total-ordering service which is responsible for all replication management (see, e.g., [Pow96, CKV01] for good surveys). Third, it allows support for Byzantine fault tolerance to be easily incorporated into the system by employing an NR-arbitrary agreement protocol of Chapter 7 combined with Masking Quorum systems [MR98] and response voting. Finally, limiting redundancy only to the places where it is really needed (namely, object replication) results in an infrastructure with only a few necessary components (cf. Section 8.4) thus simplifying the system deployment and reducing its code complexity. (Our

CORBA implementation uses 4K lines of Java code for each of the client and server implementations).

## 8.2 The Chapter Outline

This chapter is structured as follows: Section 8.3 describes the storage-centric methods we employ. Section 8.4 describes the overall design of *Aquarius*, and details of the implementation are provided in Section 8.5. Section 8.6 presents performance measurements of *Aquarius*. Section 8.7 describes a replicated database built using *Aquarius*. Section 8.8 outlines possible future developments.

## 8.3 Replication methodology

Our methodology for supporting consistent, universal object replication utilizes the agreement protocol of Chapter 4. The stability assumptions are encapsulated into a separate leader election module.

For the sake of presentation, we assume the NR-crash failure model and consider a single application object replicated at $n > 2t$ servers up to $t$ of which can crash. We make use of an RPC communication facility that supports asynchronous invocations and guarantees that an operation issued by a correct client eventually reaches all its correct targets. The algorithm tolerates any number of client failures. We first give a brief description of the ordering algorithm. We then outline the implementation of leader election.

### 8.3.1 Operation ordering

The operation ordering is carried out by the client side of the algorithm whose pseudocode is depicted in Figure 8.7 in the appendix. The client utilizes replicated servers for storing application requests and ordering decisions. The implementation employs two separate threads: one for disseminating application requests to the servers (the *dissemination* thread), and the other one for ordering previously submitted requests (the *ordering* thread). At the core of the ordering thread is an agreement protocol similar to that described in Chapter 4 with the decision value being an ordering of operations, represented by a sequence of operation identifiers (prefix). In the worst case, the algorithm invokes three communication phases: The first phase is used to discover the latest decision value which is then extended with newly submitted operations to obtain the next decision value. This new value is then proposed and committed to the servers. The clients employ unique ranks (similar to the Paxos ballots) to prevent concurrent leaders from proposing conflicting decisions and to reliably determine a decision value to extend. We note that the actual implementation employs several optimizations that reduce the number of communication rounds in failure free runs to one.

To ensure progress, the ordering thread employs a simple backoff based probabilistic mutual exclusion mechanism similar to that of [CMR01][2] that works as follows: Whenever an ordering attempt fails because of an intervening ordering attempt by a higher ranked client (PROPOSE returns *nack*), the ordering thread backs off and then repeats its ordering attempt. The backoff period is chosen randomly from the interval $[\Delta, \Delta f(attempt)]$, where $\Delta$ is the upper bound on the time required to order a single operation, *attempt* counts the number of unsuccessful ordering attempts, and $f$ is a function monotonically increasing over *attempt* (e.g., $f = 2^{attempt}$ for exponential backoff). By choosing backoff times from monotonically increasing intervals, the method ensures exclusion among a priori unknown (but eventually bounded) number of simultaneously contending clients. Note however, that the mechanism does not guarantee starvation freedom if the clients keep submitting new operations. The Aquarius implementation overcomes this problem by introducing persistent client side agents (proxies) and extending the basic backoff protocol to support long-lived leader election.

The server side of the ordering algorithm (see Figure 8.8 in the appendix) is very simple: It supports only three operations: GET to read the replica's state; PROPOSE to store a possibly non-final ordering proposal whose rank is the highest so far; COMMIT to finalize the order and apply the operations to the application object.

## 8.4 System Architecture

### 8.4.1 Overview

Our implementation of the storage-centric approach forms a middle-tier of client-proxies, whose role is to act on behalf of client requests. Proxy modules may be co-located with client processes, but need not be so. There are several reasons for logically separating between clients and proxies.

**Persistence:** Proxies can be administered with prudence, and thus a proxy that becomes a leader of the ordering protocol may prevail for a long period. This entails considerable savings in the bootstrap of the ordering scheme. In contrast, clients might enter and leave the system frequently, generating contention for the leadership position.

[2]The alternative was to use a deterministic algorithm of Chapter 5. Evaluating and comparison of the probabilistic and deterministic approaches is the subject of the ongoing work.

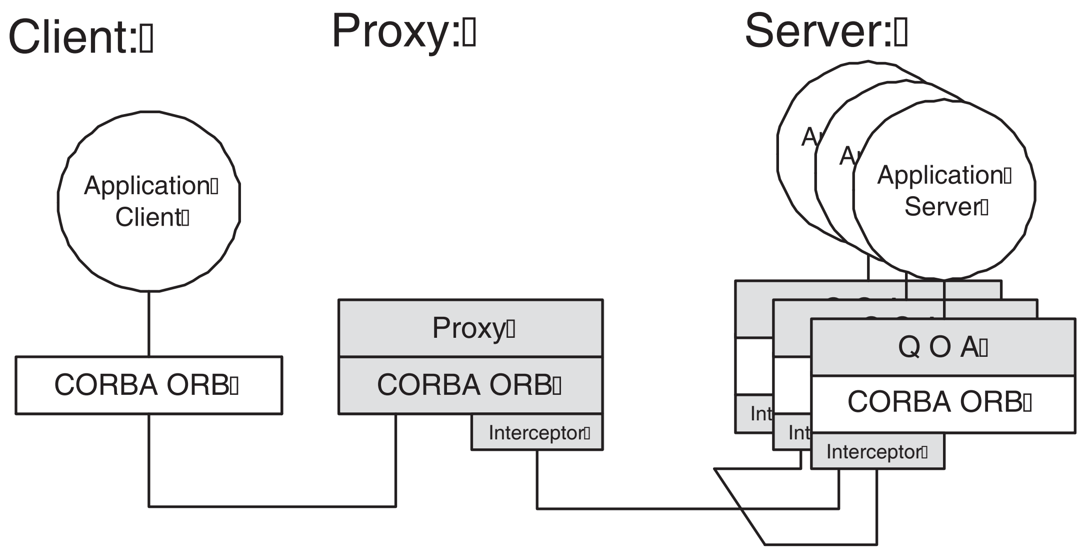

Figure 8.1: The *Aquarius* Architecture

**Efficiency:** A proxy may serve multiple clients, as well as multiple objects. Several important optimizations result from this (as described in more detail below), e.g., batching dissemination requests and ordering operations.

**Transparency:** The client code requires little or no change. For legacy applications this means that they will simply use the remote CORBA reference as they did before, without any changes.

**Extendibility:** The proxy is an ideal location for extended functionality, as it is in the critical path of the protocol. For example, we envision running in the future monitoring and profiling tools on the distributed application.

On the server side, the storage-centric approach requires simple functionality. This simplicity allows us to use the *Object Adapter* approach, introduced in [FH02]. CORBA defines an object adapter, called the *Portable Object Adapter (POA)*, for use in client-server applications. This adapter can be extended and customized according to the application's specific needs. Aquarius defines the *Quorum Object Adapter (QOA)* which adds the functionality required by the ordering protocol without modifying the application server code.

On the client side, each request is assigned a unique ID used in the ordering protocol. This unique ID is added transparently using the CORBA Portable Interceptor mechanism. This request is sent to the proxy and forwarded to the application servers. No change is

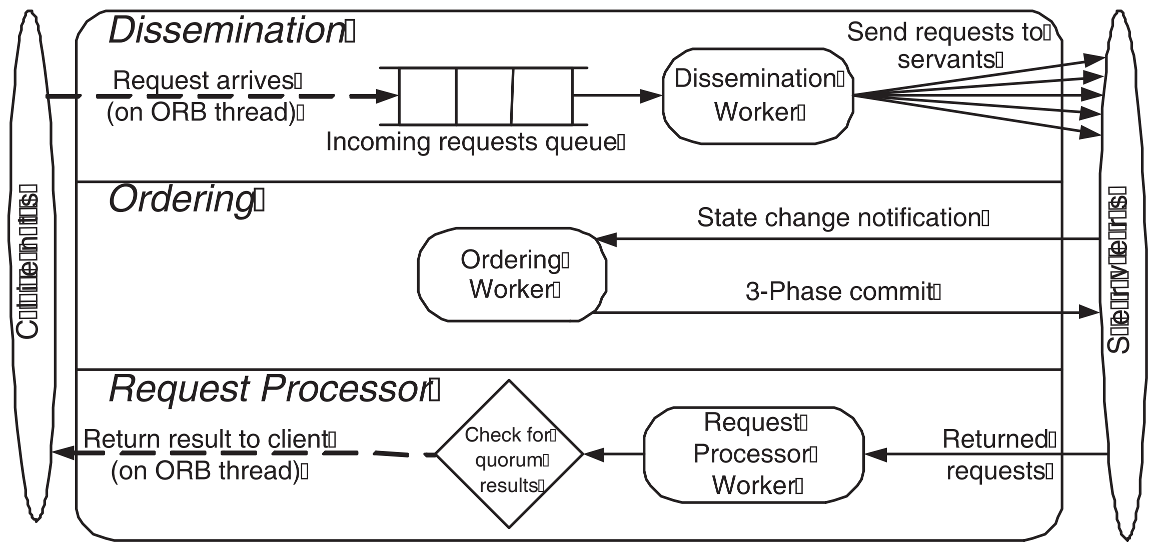

Figure 8.2: The *Aquarius* Proxy Architecture

required in the client's code.

These then are the main components of the Aquarius system: the QOA that supports the additional server functionality, the proxy that handles dissemination and ordering, and Portable Interceptors that transparently transfer the unique IDs. This architecture is in the spirit of the storage-centric approach, maintaining state on the servers and executing the protocol on the proxy, while enjoying the advantages of a three-tier architecture requiring minimal changes to existing code.

### 8.4.2 Proxy

The *Aquarius* proxy is a stateless server - it holds no persistent data, and requires no stable storage. This allows a backup proxy to assume the leadership position in case of a proxy failure without reconfiguration. The new leader automatically acquires all relevant information in the process of executing the ordering protocol.

The proxy consists of two parallel threads of execution, each with its own separate data and interface. The first, called the *dissemination thread*, is responsible for disseminating client requests to all replica servers, and to process their responses. The second thread, called the *ordering thread*, is responsible for creating a total order of all client requests. The proxy receives client requests via the CORBA Dynamic Skeleton Interface (DSI) which allows it to receive client calls from any application. The message is then forwarded asynchronously via the CORBA Dynamic Invocation Interface (DII) to all replica servers. Once the results from the replicas return, the proxy returns the agreed upon result to the client.

CORBA Object Request Brokers (ORBs) commonly support two threading paradigms: THREAD-PER-REQUEST and THREAD-POOL. In the THREAD-PER-REQUEST model, a new thread is created to dispatch each request. In the THREAD-POOL model, a pool of threads is allocated during bootstrap. For each incoming request a thread is removed from the pool and is used to dispatch the request. Once the request is completed, the thread returns to the pool. If no thread is available for a request, it is queued until a thread is ready. Both these models are inadequate for the tasks required of the Aquarius proxy. Since requests can block until they are ordered a THREAD-POOL may easily be exhausted, halting all future operations in the proxy. The THREAD-PER-REQUEST model is not affected by this problem, but it does not scale well. Therefore a different approach is employed.

The Aquarius proxy uses a fixed number of threads which run fast, non-blocking operations. All incoming requests are queued and dispatched by the dissemination thread. Ordering requests are sent and received by the ordering thread, and a third thread, called the *request processor*, is responsible for collecting replica responses and returning results to the clients. This model uses minimal resources while ensuring the proxy does not halt.

The Aquarius system can manage any number of replicated objects, where each object can have any number of replicas, and be accessed by any number of proxies. For each object, one of the participating proxies is designated as the leader proxy that runs the ordering protocol for that replication group. The other proxies are considered followers, and implicitly rely on the leader to order their requests.

### 8.4.3 Quorum Object Adapter (QOA)

In a standard, non fault-tolerant CORBA application, the server object factory instantiates an implementation object and activates it in a CORBA Portable Object Adapter (POA). It then publishes the reference to this object. In *Aquarius*, the object factory must activate the object in a QOA. This is the only change required in the server. Note that the logic of the implementation remains unchanged. Only the factory class, which is usually a separate and much simpler module, is changed.

The QOA acts as a wrapper to the implementation object and an additional CORBA object, which is responsible for handling the server side of the total order protocol. This additional object simply maintains the ordering state of the replica, and changes it according to the ordering calls. It is the simplicity of these calls that allows us to embed them in an object adapter.

Dispatching of client calls in the QOA is handled using the CORBA *RequestDispatcher* mechanism: ordering operations are dispatched by the ordering object, and any other operation is kept in storage until they are ready to be executed by the application server object.

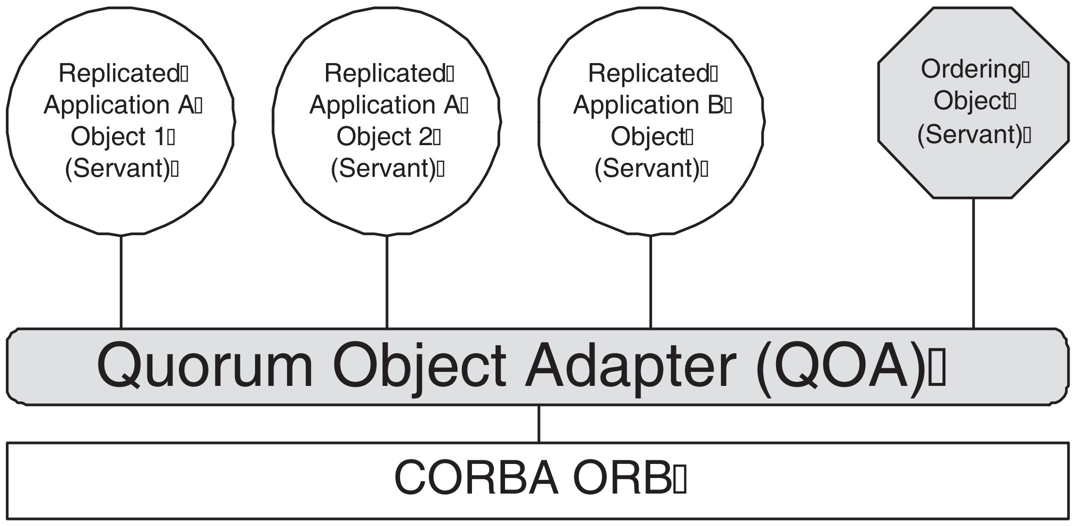
Replicated Application A Object 1 (Servant)
Replicated Application A Object 2 (Servant)
Replicated Application B Object (Servant)
Ordering Object (Servant)
Quorum Object Adapter (QOA)
CORBA ORB

Figure 8.3: The *Aquarius* Quorum Object Adapter Architecture

Operations are ready to be executed once their unique id is committed by the QOA and there is no other unique id prior to it in the total order which has not been executed.

Note that the execution of operations is delegated to a separate thread, so as not to block the ordering object from processing further ordering requests.

## 8.5 Implementation

### 8.5.1 Bootstrap

An *Aquarius* proxy is responsible for creating or accessing object replicas. These replicas can be created by a call from the proxy, in which case they are created within the context of an *Aquarius* QOA or they can be existing replicas, created previously by an *Aquarius* proxy. The proxy then creates an object group reference, which looks like a standard CORBA reference, and that can used by any client.

All bootstrap operations use the interfaces defined in the FT-CORBA specification [Obj00]. Specifically, the *GenericFactory* interface is used for creating replicas and groups, and the *PropertyManager* interface is used for specifying configuration parameters.

Once replicas are created and a group reference exists for them, the proxy is responsible for disseminating any requests made on the object group reference to all participating replicas. Any number of proxies can access the object replica-group. Each proxy creates its own object group reference, but all use the same group of replicas. One of these proxies is considered the leader, and is responsible for executing the ordering protocol. The proxies are independent of each other, and are unaware of any other proxies in the system.

### 8.5.2 Ordering

The ordering protocol is a three phase commit protocol, as defined in [CMR01]. The parameters of each phase in the protocol are arrays of unique IDs which detail the total order of the operations, and pointers to the current position of the total order.

Forming the ordering on message IDs is an optimization of the storage-centric approach, as it decouples the dissemination and ordering messages. Each message is assigned a unique ID. The ordering protocol does not require the actual message contents, only its ID, serving to minimize the size of the messages in the ordering protocol. In Aquarius this is supported transparently using CORBA *Portable Interceptors*, which add the unique ID to the message header without any change to the application's client or server code.

A asynchronous notification mechanism between servers and proxies allows the proxy ordering thread to be idle most of the time. As request messages arrive at the replicas, they generate a state change in the server QOA, which notifies one proxy – the current presumed leader – of the change. This initiates the ordering protocol at the proxy leader. Once the message has been committed at the QOA, it is dispatched to the application code, which then executes the request and returns its reply to the proxy that sent it.

In order to support these notifications without registering the proxy at each of the QOAs, a proxy leader calls a remote operation on each of the replicas which only returns if a request is pending at this QOA. If no such operation is pending, the request blocks until a request arrives. This operation is equivalent to the `get()` operation of the ordering protocol, except that it blocks until it can return useful information. The additional remote call adds very little overhead at each of the QOAs while saving the bandwidth required for continuous execution of the ordering protocol.

### 8.5.3 Handling Proxy Failures

In order to ensure fault-tolerance all components in the system must be replicated, and the Aquarius proxy is no exception. With multiple proxies in place, a client can overcome a proxy failure by simply switching to any other proxy which is a client for the object, or by instructing a proxy to join this group (this can be implemented transparently with client-side Portable Interceptors, as described in [FH02]). The situation is more complicated if the proxy that fails is also the leader. In this case, ordering operations stops and the QOAs will not execute any application code. Therefore a new leader must be elected.

A proxy that is part of a group can suspect that the proxy leader for the group has failed if a user-defined timeout has expired since it disseminated a client request to the object

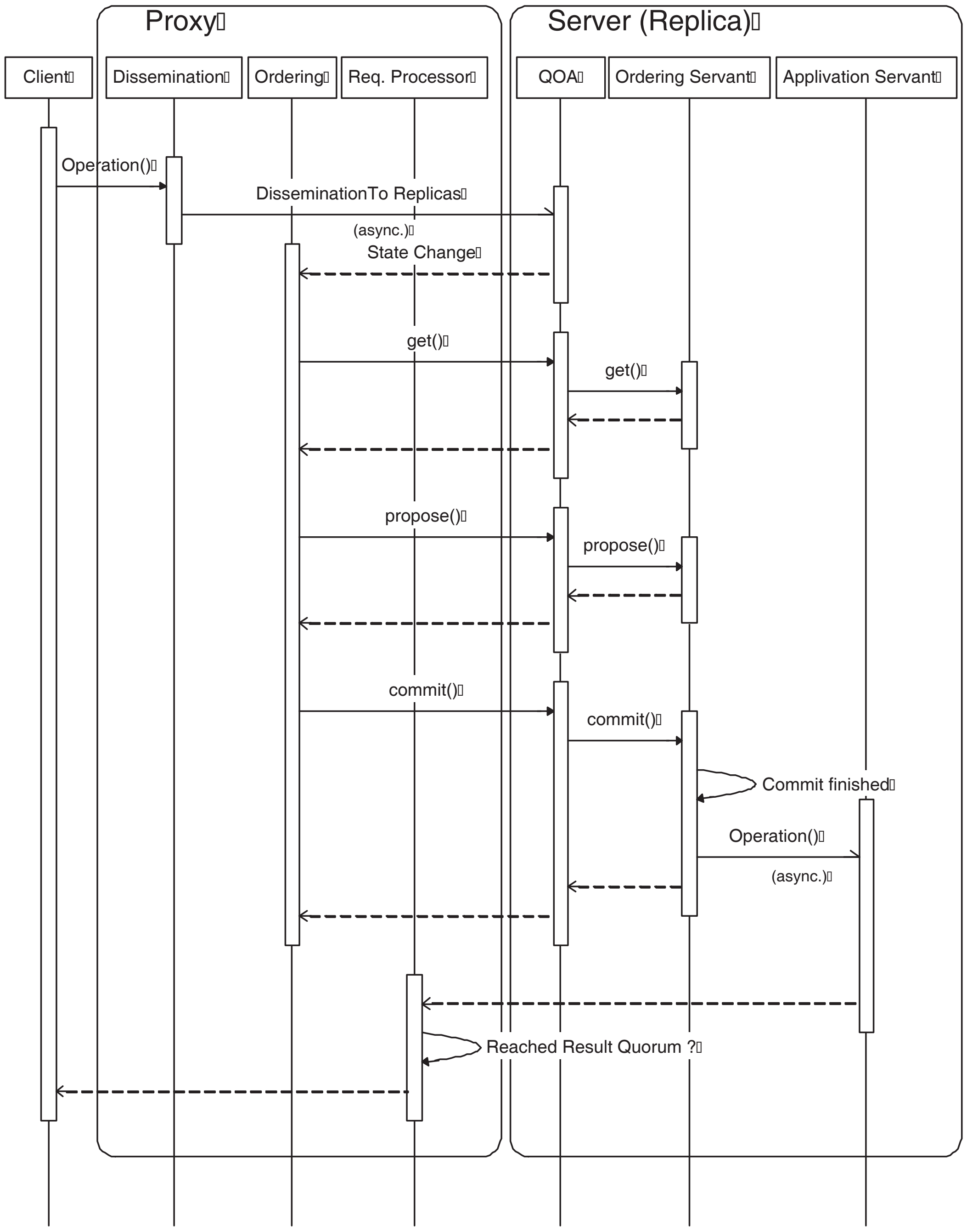

Figure 8.4: The *Aquarius* Sequence Diagram

replicas. All proxies that suspect such a failure will attempt to become leaders by executing the ordering protocol. Only one will succeed, and it becomes the new leader, while the others will fail on a *RankException* and remain followers. The statelessness of the ordering protocol makes this possible, since a new leader can simply resume where the last one failed.

### 8.5.4 Summary of optimizations and enhancements

While implementing the system we found some practical improvements of value:

**Garbage collection:** Maintaining and transmitting the state of the ordering protocol over time requires a growing amount of resources. *Aquarius* minimizes this by adding the notion of a 'stable line'. This in the most recent command that has been executed by *all* servers. The state before this request can be discarded. This saves considerable amounts of information that must be sent between the proxy and the QOAs.

**Message batching:** The proxy can batch the ordering of multiple messages, sent by multiple clients in one ordering message, thus saving the bandwidth and round trip time required for the remote calls.

**Notifications:** The notifications described above are a practical solution required for the proxy architecture, which also increases the efficiency of the system.

**Threading model:** The specially designed threading model described above requires a constant amount of memory allowing for greater scalability, while ensuring that the proxy never blocks.

## 8.6 Performance

This section outlines measurements for the *Aquarius* system in a test environment. The system was implemented using ORBacus 4.1.0 [ION] and the Java language, using JDK 1.3.1. The experiments were performed on Pentium III PCs over a 100Mbps local area network. Each PC is equipped with a 500Mhz CPU, 256MB of RAM, and runs the Debian Linux (kernel 2.4.18) operating system.

A simple client/server was developed for the experiments. The server contains a single remote operation that receives a buffer of varying size as a parameter. This allows us to measure the round trip time of a request as affected by the size of the request. Two sets of tests were run: the first tests the performance of the system with one client and proxy, and an increasing number of application servers (equivalently, the replication degree). The second test increases the number of concurrent clients while working with a single proxy and five application servers. The results of both tests are depicted in Figure 8.5 and Figure 8.6, respectively.

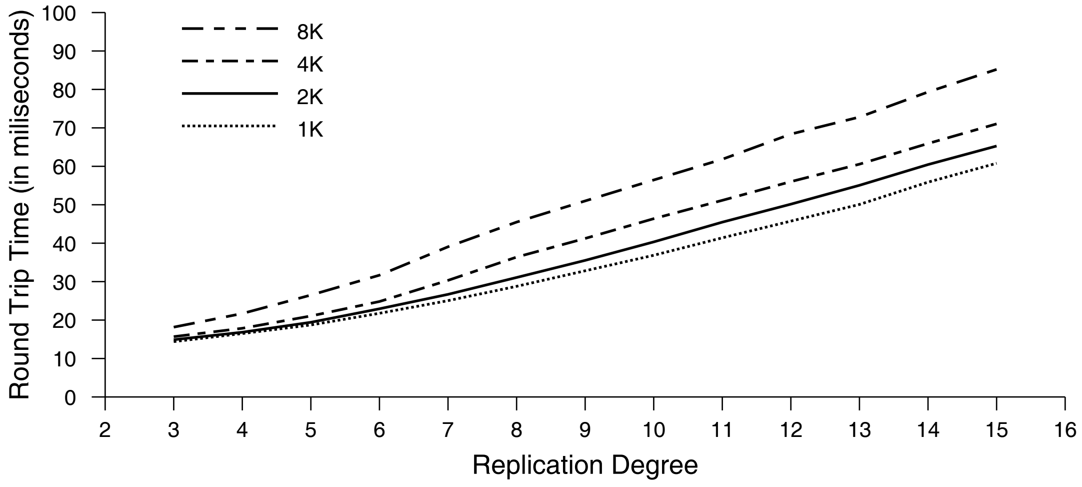

Figure 8.5: Round trip time according to replication degree and request size

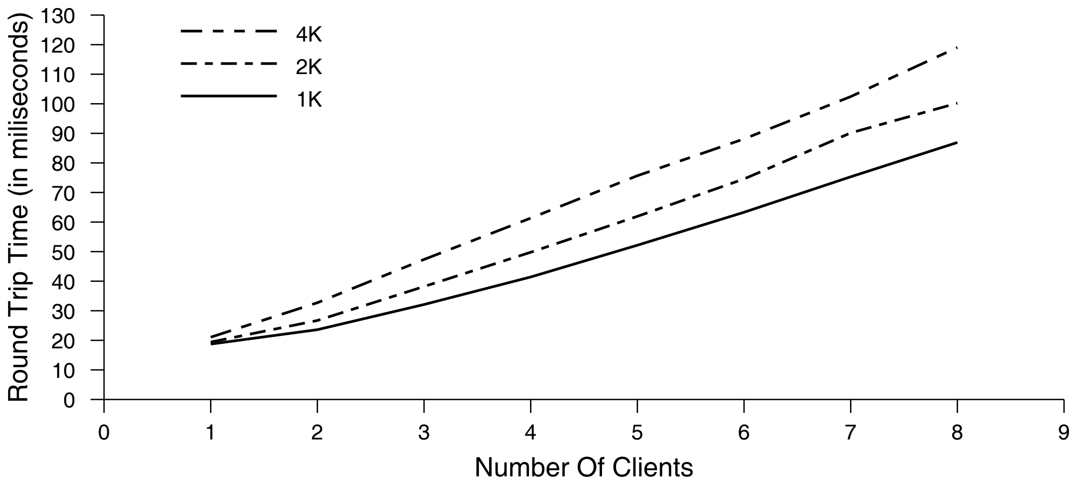

Figure 8.6: Round trip time according to number of clients and request size (Using one proxy and 5 application servers)

## 8.7 A database application

Recent years witnessed a great interest in replicated systems, databases in particular. The need to maintain the availability of commercial data led to the development of several database replication techniques. All major database vendors support some sort of replication for their products. Other companies offer middleware that enables replication.

Our approach offers an easy-to-construct replication middleware. The combination of such a middleware with a standard, non-replicated, database is a cheap alternative to commercial products.

We have built a prototype replicated database using our methods over the HSQL [HSQ] database, an open source relational database system written in Java. HSQL is a JDBC-compliant SQL database, but has no replication support. By combining HSQL with the Aquarius system, replication is achieved with very little additional code. A simple server object was written in Java, that supports a single remote operation. This operation receives an SQL query, and returns its result. The server itself requires only two hundred lines of code, including server initialization and configuration. The resulting system shows good scalability, and supports 50 operations per second for 5 replicas on the test environment described above.

## 8.8 Future Directions

Future work on the *Aquarius* system includes several possible directions.

**Quorum definitions:** The pluggable quorum management module allows new quorum systems to be tested for efficiency. In addition, the module responsible for communicating with quorums may be extended beyond strict quorums access. Two possible examples is allowing asynchronous backup of a slower secondary site (without slowing the primary site) or supporting dirty reads (read operations that do not require a quorum of replies, risking reading old information).

**Recovery:** Transferring the state of an object replica-group to a new replica, or to a faulty one that recovers.

**Monitoring and Security:** Proxies are an excellent location for handling system monitoring and maintaining access control lists. Portable Interceptors can be used to add these features transparently to the system.

## 8.9 Pseudo Code Of The Ordering Protocol

```
Boolean $finish = false$;
Dissemination thread:
    When an operation $op$ is submitted for ordering:
        Assign $op$ a unique id;
        Invoke $\textsc{submit}(id, op)$ on all servers;
        Wait until some server responds with $\langle id, res \rangle$;
        $finish \leftarrow true$;
        Return $res$;

Ordering thread:
    do
        Wait until $(isLeader \vee finish)$;
        While $(isLeader \wedge \neg finish)$ do
            Pick a unique, monotonically increasing rank $r$;
            Invoke $\textsc{get}(r)$ on $n$ servers;
            Wait for more than $n/2$ servers $s_i$ to respond with $\langle r_i, \mathsf{prefix}_i, \mathsf{pending}_i \rangle$;
            Let $Pending = \bigcup_i \mathsf{pending}_i$;
            Let $\mathsf{prefix} = \mathsf{prefix}_j$ such that $r_j = max_i r_i$;
            For each $id \in Pending$ which is not included in $\mathsf{prefix}$
                $\mathsf{prefix} \leftarrow append(\mathsf{prefix}, id)$;
            Invoke $\textsc{propose}(r, \mathsf{prefix})$ at all servers;
            Wait for more than $n/2$ servers $s_i$ to respond with $ack/abort$;
            If more than $n/2$ servers respond with $ack$
                Invoke $\textsc{commit}(r, \mathsf{prefix})$ on all servers;
        od
    Until $(finish)$
```

Figure 8.7: Storage-centric operation ordering: The client side

```
Sets $\mathsf{pending}$, $\mathsf{Ops}$, initially empty;
Sequences $\mathsf{prefix}^p$, $\mathsf{prefix}^c$, initially empty;
Ranks $\mathsf{getRank}$, $\mathsf{propRank}$, initialized to a predefined initial value;

SUBMIT$(id, op)$:
    $pending \leftarrow \mathsf{pending} \cup \{id\}$;
    $\mathsf{Ops} \leftarrow \mathsf{Ops} \cup \{\langle id, op\rangle\}$;
    Execute WAITANDAPPLY$(id)$
    in a separate thread;

WAITANDAPPLY$(id)$:
    Wait until:
      (1) $id$ appears on $\mathsf{prefix}^c$;
      (2) all operations preceding $id$
            in $\mathsf{prefix}^c$ were applied;
      (3) $\langle id, op\rangle \in \mathsf{Ops}$ for some operation $op$;
    Apply $op$ to the application object
    and return the result to client;

GET$(r)$:
    if $(r > \mathsf{getRank})$
        $\mathsf{getRank} \leftarrow r$;
    return $\langle \mathsf{propRank}, \mathsf{prefix}^p, \mathsf{pending}\rangle$;

PROPOSE$(r, \mathsf{prefix})$:
    if $(getRank \leq r \vee \mathsf{propRank} < r)$
        $\mathsf{propRank} \leftarrow r$;
        $prefix^p \leftarrow \mathsf{prefix}$;
        return $ack$;
    return $nack$;

COMMIT$(r, \mathsf{prefix})$:
    if $(\mathsf{propRank} \leq r)$
        $prefix^c \leftarrow \mathsf{prefix}$;
    return $ack$;
```

Figure 8.8: Storage-centric operation ordering: The server side

# Chapter 9

# Conclusions

This dissertation has presented the results of our investigation into the possibility and cost of building fault-tolerant services in storage-centric systems. We have conducted our study within a precise mathematical framework given by an asynchronous shared memory model with objects prone to non-responsive (NR) failures. We have considered two types of shared object failures: non-responsive crash (NR-crash) and non-responsive arbitrary (NR-arbitrary) failures. The first failure mode models benign faulty storage servers, whereas the second one models arbitrary, possibly malicious, storage failures.

Our focus has been on solving the Consensus problem. We have shown solutions for Consensus in storage-centric systems subject to benign and Byzantine storage failures under the assumption of unreliable failure detector of class $\Omega$. Our solutions are based on a novel ranked register object that promotes understanding and analysis of Paxos and of general coordination in distributed systems. We have also shown an implementation of a failure detector of class $\Omega$ in a partially synchronous shared memory model, thus providing a complete solution to the Consensus problem under realistic environment assumptions. Both our consensus and failure detector implementations are oblivious to the number of participating client processes.

We have investigated the cost of implementing wait-free objects out of of $n > 3t$ Byzantine fault-prone storage servers. To this end, we have established a tight lower bound on the round complexity of wait-free read-write register implementations resilient to $t < n/3$ base object failures (optimal resilience). We have also demonstrated that the implementation round complexity increases significantly when the failure resilience increases from $t < n/4$ to $t < n/3$.

Finally, we have demonstrated a practical value of the storage-centric design paradigm by utilizing it for implementing a CORBA fault-tolerance infrastructure, called Aquarius.

For the future, we believe that fault-tolerance in storage-centric model will continue to be a fruitful research area for both theoreticians and practitioners alike. From the theoretical standpoint, an interesting research direction will be to extend the Byzantine fault-tolerant Consensus solutions to support malicious client failures. Also, the relationship between the storage-centric and message-passing models is not yet fully understood. In this respect, an interesting research problem will be to establish a correspondence between the pure message-passing and the storage-centric models that will allow the upper and lower bounds to be carried over between the models. From the practical perspective, it will be interesting to investigate the applicability of the storage-centric paradigm in new application domains, such as today's peer-to-peer systems. Another interesting direction with both theoretical and practical flavors will be to enhance storage-centric systems to support atomic transactions involving multiple replicated object instances.

# Bibliography

[AAT97] Rajeev Alur, Hagit Attiya, and Gadi Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM Journal on Computing*, 26(2):539–556, April 1997.

[ABO03] Hagit Attiya and Amir Bar-Or. Sharing memory with semi-byzantine clients and faulty storage servers. In *22nd Symposium on Reliable Distributed Systems (SRDS)*, October 2003.

[ADGFT03] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega with weak reliability and synchrony assumptions. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, July 2003.

[ADN$^+$96] Thomas E. Anderson, Michael Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[AGG00] Khalil A. Amiri, Garth A. Gibson, and Richard Golding. Highly concurrent shared storage. In IEEE, editor, *Proceedings of* 20th *International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 298–307, Taipe, Taiwan, R.O.C, 2000. IEEE Computer.

[AGMT95] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the Association of the Computing Machinery*, 42:1231–1274, 1995.

[AL94] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[Asp03] James Aspnes. Randomized protocols for asynchronous consensus. *Distrib. Comput.*, 16(2-3):165–175, 2003.

[AT96a] Rajeev Alur and Gadi Taubenfeld. Contention-free complexity of shared memory algorithms. *Information and Computation*, 126(1):62–73, 1996.

[AT96b] Rajeev Alur and Gadi Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1):1–10, 1996.

[AT99] Rajeev Alur and Gadi Taubenfeld. How to share a data structure: A fast timing-based solution. In *5th IEEE Symposium on Parallel and Distributed Processing*, pages 470–477, 1999.

[AUS98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII), ACM SIGPLAN*, pages 81–91. ACM SIGARCH v26/SIGOPS v32 n5/SIGPLAN v 33 n 11, November 1998.

[Baz00] Rida A. Bazzi. Synchronous byzantine quorum systems. *Distributed Computing*, 13(1):45–52, 2000.

[BDFG01] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Deconstructing paxos. Technical Report DSC ID:200106, École Polytechnic Fédérale de Lausanne (EPFL), January 2001.

[BDFG03] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Deconstructing paxos. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 34, 2003.

[BDG02] Romain Boichat, Partha Dutta, and Rachid Guerraoui. Asynchronous leasing. In *7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, San Diego, California, January 2002. Invited paper.

[BJ87] Ken P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the* $11^{th}$ *ACM Symposium on OS Principles*, pages 123–138, Austin, TX, USA, 1987. ACM SIGOPS, ACM.

[BL93] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[BM03] Roberto Baldoni and Carlo Marchetti. Three-tier replication for ft-corba infrastructures. *Software Practice and Experience*, 33:767–797, May 2003.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[Bur00] Randal Burns. *Data management in a distributed file system for Storage Area Networks.* PhD thesis, Department of Computer Science, University of California, Santa Cruz, March 2000.

[CD89] Benny Chor and Cynthia Dwork. Randomization in Byzantine agreement. In *Advances in Computing Research 5: Randomness and Computation*, pages 443–497. JAI Press, 1989.

[CF99] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), 1999.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[CKM03] Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. The inherent cost of optimal resilience wait-free storage from byzantine components. Submitted for publication, August 2003.

[CKV01] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 4(33):1–43, December 2001.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[CM02] Gregory V. Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. In *Proceedings of the twenty-first Annual Symposium on Principles of Distributed Computing (PODC-02)*, pages 78–87, New York, July 2002. ACM Press.

[CM03] Gregory V. Chockler and Dahlia Malkhi. Light-weight leases for large-scale coordination. Submitted for publication, August 2003.

[CMD03] Gregory V. Chockler, Dahlia Malkhi, and Danny Dolev. A data-centric approach for scalable state machine replication. In A. Schiper, A. A. Shvartsman,

H. Weatherspoon, and B. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 159–163. Springer-Verlag, 2003.

[CMMR03] Gregory V. Chockler, Dahlia Malkhi, Barak Merimovich, and David Rabinowitz. Aquarius: A Data-Centric approach to CORBA fault-tolerance. In *The Workshop on Reliable and Secure Middleware (WRSM), in the International Conference on Distributed Objects and Applications (DOA)*, Sicily, Italy, November 2003. To appear.

[CMR01] Gregory V. Chockler, Dahlia Malkhi, and Michael K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-01)*, pages 11–20, Los Alamitos, CA, April 2001. IEEE Computer Society.

[Con] National Storage Industry Consortium. `http://www.nsic.org/nasd`.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronization needed for distributed consensus. *JACM*, 34(1):77–97, January 1987.

[DKKV03] Roberto Segala Dilsun K. Kaynar, Nancy Lynch and Frits Vaandrager. Timed i/o automata: A mathematical framework for modeling and analyzing real-time systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS 2003)*, pages 166–177, Cancun, Mexico, December 2003. IEEE Computer Society.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[Fel98] Pascal Felber. *The CORBA Object Group Service*. PhD thesis, École Polytechnic Fédérale de Lausanne (EPFL), 1998.

[FGS98] Pascal Felber, Rachid Guerraoui, and André Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[FH02] Roy Friedman and Erez Hadad. Fts: A high-performance corba fault-tolerance service. In *The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 61–68, 2002.

[FHS98] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FLS01] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, 2001.

[GC89] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, Arizona, December 1989.

[GGT97] Howard Gobioff, Garth A. Gibson, and Doug Tygar. Security for network attached storage devices. Technical Report CMU-CS-97-185, CMU, October 1997.

[GL03] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, February 2003.

[GM02] Eli Gafni and Michael Mitzenmacher. Analysis of timing-based mutual exclusion with random times. *SIAM Journal on Computing*, 31(3):816–837, June 2002.

[GMT01] Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'2001)*, August 2001.

[GNA$^+$97] Garth Gibson, David Nagle, Khalil Amiri, Fay Chang, Howard Gobioff, Erik Riedel, David Rochberg, and Jim Zelenka. Filesystems for network-attached secure disks. Technical Report CMU-CS-97-112, CMU, March 1997.

[GNA$^+$98] Garth Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective high-bandwidth storage architecture. In *Proceedings of the 8th*

*International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII), ACM SIGPLAN*, pages 92–103. ACM SIGARCH v26/SIGOPS v32 n5/SIGPLAN v 33 n 11, November 1998.

[GV04] Rachid Guerraoui and Marko Vukolic. Private communication and manuscript in progress. 2004.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HMF98] Steve Hotz, Rodney Van Meter, and Gregory Finn. Internet protocols for network-attached peripherals. In Ben Kobler, editor, *Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in Cooperation with Fifteenth IEEE Symposium on Mass Storage Systems*, March 1998.

[HMS99] John H. Hartman, Ian Murdock, and Tammo Spalink. The swarm scalable storage system. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 74–81. IEEE, May 1999.

[HSQ] Hsql database. `http://www.hsqldb.org`.

[ION] IONA. *ORBacus*. `http://www.iona.com/products/orbacus_home.htm`.

[ION94] IONA Technologies Ltd. and Isis Distributed Systems, Inc. *IONA and Isis. An Introduction to Orbix+ISIS*, 1994.

[JCT98] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, May 1998.

[JTT00] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000.

[KD00] Idit Keidar and Danny Dolev. Totally ordered broadcast in the face of network partitions. In D. Avresky, editor, *Dependable Network Computing*, chapter 3. Kluwer Academic Publications, 2000.

[KLSV04] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits Vaandrager. The Theory of Timed I/O Automata. Technical Report MIT-LCS-TR-917a, MIT Laboratory for Computer Science, Cambridge, MA, apr 2004. Manuscript in progress.

[KR03] Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85:47–52, January 2003.

[LAA87] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, editor, *Advances in Computing Research, Parallel and Distributed Computing*, volume 4, pages 163–183. JAI Press, Inc., Greenwich, Conn., 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam86] Leslie Lamport. On interprocess communication, Parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[Lam96] Butler W. Lampson. How to build a highly available system using consensus. In Babaoglu and Marzullo, editors, *10th International Workshop on Distributed Algorithms (WDAG 96)*, volume 1151 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, Berlin Germany, 1996.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam01a] Leslie Lamport. Paxos made simple. *Distributed Computing Column of SIGACT News*, 32(4):34–58, December 2001.

[Lam01b] Butler W. Lampson. The ABCD's of paxos (1 page). In *PODC: 20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Newport, Rhode Island, USA, August 2001. Lamport Celebration Lecture 2.

[LAV01] Subramanian Lakshmanan, Mustaque Ahamad, and H. Venkateswaran. A secure and highly available distributed store for meeting diverse data storage needs. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 251–260. IEEE, July 2001.

[LH94] Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *8th International Workshop on Distributed Algorithms (WDAG)*, volume 857 of *Lecture Notes in Computer Science (LNCS)*, pages 280–295. Springer-Verlag, Berlin, Germany, 1994.

[LM97] Sean Landis and Silvano Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[LS92] Nancy A. Lynch and Nir Shavit. Timing-based mutual exclusion. In Robert Werner, editor, *Proceedings of the Real-Time Systems Symposium - 1992*, pages 2–11, Phoenix, Arizona, USA, December 1992. IEEE Computer Society Press.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LT89] Nancy Ann Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LT96] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Computer Architecture News*, pages 84–93. ACM SIGARCH/SIGOPS/SIGPLAN, October 1996.

[MAD02] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)*, Lecture Notes in Computer Science (LNCS), Toulouse, France, October 2002. Springer-Verlag, Berlin, Germany.

[Mal02] Dahlia Malkhi. From byzantine agreement to practical survivability. In *International Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS 2002)*, Osaka, Japan, October 2002.

[MMSN98] Louise E. Moser, P. Michael Melliar-Smith, and Priya Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MR00] Dahlia Malkhi and Michael K. Reiter. An architecture for survivable coordination in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, March/April 2000.

[MT00] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *14th International Symposium on Distributed Computing (DISC'2000)*, volume 1914 of *Lecture Notes in Computer Science*, pages 164–178. Springer-Verlag, Berlin Germany, 2000.

[Obj99] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.3 edition, June 1999.

[Obj00] Object Management Group. *Fault Tolerant CORBA Specification*, ptc/2000-04-04 edition, April 2000.

[PLL00] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243(1–2):35–91, July 2000.

[Pow96] David Powell, editor. *Group communication*, volume 39(4), April 1996.

[RFGN01] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[Ske81] Dale Skeen. Nonblocking commit protocols. In Y. Edmund Lien, editor, *Proceedings of the 1981 International Conference on Management of Data*, pages 133–142, Ann Arbor, Michigan USA, 1981. ACM SIGMOD, New York.

[Ske82] Dale Skeen. *Crash Recovery in a Distributed Database System.* Ph.D. thesis, University of California, Berkeley, May 1982.

[SMS$^+$03] Julian Satran, Kalman Meth, Costa Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. *iSCSI.* IP Storage Working Group, IETF, draft-ietf-ips-iscsi-20 edition, January 2003. Internet draft.

[SPW03] Cheng Shao, Evelyn Pierce, and Jennifer Lundelius Welch. Multi-writer consistency conditions for shared memory objects. In *17th International Symposium on Distributed Computing (DISC'2003)*, 2003.

[TML97] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, pages 224–237, New York, USA, October 1997. ACM Press.

[ZSR02] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.