

# Robust Services in Dynamic Systems

by

Rodrigo Seromenho Miragaia Rodrigues

Submitted to the Department of Electrical Engineering and Computer  
Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[June 2005]

February 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author .....

Department of Electrical Engineering and Computer Science

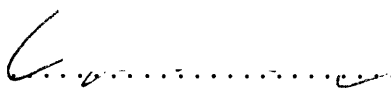
February 23, 2005

Certified by .....

Barbara H. Liskov

Ford Professor of Engineering

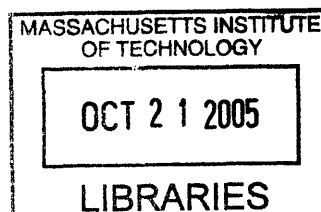
Thesis Supervisor

Accepted by .....  


Arthur C. Smith

Chairman, Department Committee on Graduate Students

**ARCHIVES**





# Robust Services in Dynamic Systems

by

Rodrigo Seromenho Miragaia Rodrigues

Submitted to the Department of Electrical Engineering and Computer Science  
on February 23, 2005, in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy in Computer Science

## Abstract

Our growing reliance on online services accessible on the Internet demands highly-available systems that work correctly without interruption. This thesis extends previous work on Byzantine-fault-tolerant replication to meet the new requirements of current Internet services: scalability and the ability to reconfigure the service automatically in the presence of a changing system membership.

Our solution addresses two important problems that appear in dynamic replicated services: First, we present a membership service that provides servers and clients in the system with a sequence of consistent views of the system membership (i.e., the set of currently available servers). The membership service is designed to be scalable, and to handle membership changes mostly automatically. Furthermore, the membership service is itself reconfigurable, and tolerates arbitrary faults of a subset of the servers that are implementing it at any instant.

The second part of our solution is a generic methodology for transforming replicated services that assume a fixed membership into services that support a dynamic system membership. The methodology uses the output from the membership service to decide when to reconfigure. We built two example services using this methodology: a dynamic Byzantine quorum system that supports read and write operations, and a dynamic Byzantine state machine replication system that supports any deterministic service.

The final contribution of this thesis is an analytic study that points out an obstacle to the deployment of replicated services based on a dynamic membership. The basic problem is that maintaining redundancy levels for the service state as servers join and leave the system is costly in terms of network bandwidth. To evaluate how dynamic the system membership can be, we developed a model for the cost of state maintenance in dynamic replicated services, and we use measured values from real-world traces to determine possible values for the parameters of the model. We conclude that certain deployments (like a volunteer-based system) are incompatible with the goals of large-scale reliable services.

We implemented the membership service and the two example services. Our performance results show that the membership service is scalable, and our replicated services perform well, even during reconfigurations.

Thesis Supervisor: Barbara H. Liskov  
Title: Ford Professor of Engineering





*In the memory of my mother.*



# Acknowledgments

This work benefited immensely from the input of various persons I had the privilege of working with. First, I would like to thank Barbara Liskov for being such a demanding advisor. I also learned a lot from working with Chuck Blake on some of the material presented in this thesis, and from being co-advised by Miguel Castro during my first two years at MIT. Robert Morris also provided useful input to some of the ideas presented in this thesis. I am thankful to the members of my thesis committee, Butler Lampson, Nancy Lynch, and Robert Morris, for giving me very useful feedback.

This thesis describes joint work with Chuck Blake, who is co-responsible the analysis in Chapter 8, and Kathryn Chen, who implemented the APSS protocol that is used by the service presented in Chapter 3. I am also grateful to the Chord team that made their code readily available and answered some of my questions, which allowed me to reuse some of that code. The analysis in Chapter 8 uses measured values for the membership dynamics of real-world systems that were collected by different studies. I am very grateful to the people in the TotalRecall project at UCSD, the Farsite project at Microsoft Research (in particular Atul Adya), Stefan Saroiu, Krishna Gummadi, and Steve Gribble at UW, and Jeremy Stribling for supplying the data collected in their studies. I also want to thank Dave Andersen for letting me use his testbed to run some experiments, and for answering every question and request with a smile.

It was a pleasure working in the Programming Methodology Group at MIT. I would like to thank all the members of my group and of the PDOS group that contributed to the quality of my work by answering my questions, having interesting discussions, and putting up with my frequent interruptions when I needed a break from work. Among the many people I was fortunate to meet here, I wanted to leave a special word for Josh Cates was always willing to help everyone, and is always going to be missed.

I am very grateful to my family and friends on both sides of the ocean for their support over the years. One of the persons that influenced me the most was my grandfather who showed me that a person's integrity is above everything, even his own personal freedom.

I want to thank João Garcia and José Carlos Monteiro for softening the impact of my transition back to Europe. Thanks to you I didn't give up on going back.

I am thankful to the Portuguese Foundation for Science and Technology and the Calouste Gulbenkian Foundation for financial support.

I am also grateful to Frank Gehry for designing a workspace with so much light, and I am ungrateful for the terrible acoustics and the exiguous men's room on our floor.

Last but not least, I want to thank Sara for being there. Always.

## Support

The author was funded by the Portuguese Foundation for Science and Technology under a Praxis XXI fellowship, and by a fellowship from the Calouste Gulbenkian Foundation. This research was also supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory, and was part of the IRIS project (<http://project-iris.net/>), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660.

# Contents

<b>1</b>	<b>Introduction</b>	<b>21</b>
1.1	Contributions . . . . .	23
1.1.1	Membership Service . . . . .	24
1.1.2	Replicated Services . . . . .	25
1.1.3	Analysis of the Cost of State Maintenance . . . . .	26
1.2	Thesis Outline . . . . .	27
<b>2</b>	<b>System Model and Assumptions</b>	<b>29</b>
2.1	Assumptions . . . . .	29
2.2	System Model . . . . .	30
2.2.1	Dynamic System Membership . . . . .	33
<b>3</b>	<b>The Membership Service</b>	<b>37</b>
3.1	Membership Service Overview . . . . .	38
3.2	Centralized Membership Service . . . . .	40
3.2.1	Membership Changes . . . . .	40
3.2.2	Propagating the System Membership . . . . .	45
3.2.3	Supplying Old Configurations . . . . .	47
3.2.4	Freshness . . . . .	48
3.3	Replicated Membership Service . . . . .	49
3.3.1	Membership Changes . . . . .	50
3.3.2	Propagating System Membership . . . . .	52
3.3.3	Leases . . . . .	53

3.3.4	Correctness of the Membership Service . . . . .	53
3.4	Reconfigurable Membership Service . . . . .	53
3.4.1	Choosing the New Membership Service . . . . .	55
3.4.2	Discovering the MS . . . . .	56
3.4.3	Signing . . . . .	56
3.4.4	State Transfer . . . . .	57
3.4.5	Epoch Transition Protocols . . . . .	59
3.4.6	Correctness Conditions of the MS . . . . .	62
3.5	Discussion . . . . .	63
3.5.1	Scalability . . . . .	63
3.5.2	Garbage Collection of Old Configurations . . . . .	64
<b>4</b>	<b>Methodology for Building Dynamic Replicated Services</b>	<b>67</b>
4.1	Methodology for Transforming Replicated Services . . . . .	68
4.1.1	Detailed Steps . . . . .	69
4.2	Correctness of the Replicated Service . . . . .	71
<b>5</b>	<b>Example I: Dynamic Byzantine Quorum Replication</b>	<b>73</b>
5.1	System Overview . . . . .	73
5.2	Object Placement . . . . .	75
5.3	Storage Algorithms . . . . .	76
5.3.1	Replicated State . . . . .	76
5.3.2	Client Protocol . . . . .	77
5.3.3	Server Protocol . . . . .	81
5.3.4	State Transfer . . . . .	82
5.3.5	Garbage-Collection of Old Data . . . . .	83
5.3.6	Skipped Epochs . . . . .	84
5.4	Correctness . . . . .	85
<b>6</b>	<b>Example II: Dynamic State Machine Replication</b>	<b>87</b>
6.1	BFT State Machine Replication Overview . . . . .	88

6.1.1	Algorithm Properties . . . . .	88
6.1.2	Algorithm Overview . . . . .	89
6.2	dBFT Algorithms in the Static Case . . . . .	94
6.2.1	Client Protocols . . . . .	95
6.3	Configuration Changes . . . . .	97
6.3.1	Delimiting Epoch Changes . . . . .	97
6.3.2	Handling Epoch Changes . . . . .	98
6.4	State Transfer . . . . .	99
6.5	Garbage Collection of Old Application State . . . . .	100
6.6	Garbage Collection of Old BFT replicas . . . . .	101
6.7	Scalability of BFT with the Number of Clients . . . . .	102
<b>7</b>	<b>Implementation</b>	<b>103</b>
7.1	Membership Service Implementation . . . . .	103
7.2	Implementation of dBQS . . . . .	104
7.3	Implementation of dBFT . . . . .	105
<b>8</b>	<b>The Cost of Membership Dynamics</b>	<b>107</b>
8.1	Basic Model . . . . .	108
8.1.1	Assumptions . . . . .	108
8.1.2	Data Maintenance Model . . . . .	108
8.2	Distinguishing Downtime vs. Departure . . . . .	111
8.2.1	Needed Redundancy: Replication . . . . .	112
8.3	Detecting Permanent Departures . . . . .	113
8.3.1	Setting the Membership Timeout . . . . .	114
8.3.2	Putting it All Together: Bandwidth Usage . . . . .	118
8.4	Erasur Coding . . . . .	120
8.4.1	Needed Redundancy and Bandwidth Usage . . . . .	121
8.4.2	Replication vs. Erasure Coding: Quantitative and Qualitative Comparisons . . . . .	122
8.5	Discussion . . . . .	125

8.5.1	Limitations of the Model . . . . .	125
8.5.2	Other Optimizations . . . . .	126
8.5.3	Hardware Trends . . . . .	127
<b>9</b>	<b>Evaluation</b>	<b>129</b>
9.1	Evaluation Infrastructure . . . . .	129
9.2	The Membership Service . . . . .	130
9.2.1	Scalability . . . . .	130
9.3	Dynamic Byzantine Quorums . . . . .	139
9.3.1	Performance with a Static Configuration . . . . .	139
9.3.2	Performance During Reconfigurations . . . . .	141
9.3.3	Impact of MS operations on Application Performance . . . . .	145
9.4	Dynamic Byzantine State Machine Replication . . . . .	145
<b>10</b>	<b>Related Work</b>	<b>151</b>
10.1	Membership Service . . . . .	151
10.1.1	Applications with Membership Changes . . . . .	151
10.1.2	Group Communication Systems . . . . .	153
10.1.3	Peer-to-Peer Systems . . . . .	155
10.2	Dynamic Replication . . . . .	157
10.2.1	Systems Related to dBQS . . . . .	157
10.2.2	Systems Related to dBFT . . . . .	158
10.3	Analysis of Membership Dynamics . . . . .	159
<b>11</b>	<b>Conclusion</b>	<b>161</b>
11.1	Summary . . . . .	161
11.2	Future Work . . . . .	163
11.2.1	Membership Service . . . . .	163
11.2.2	dBQS . . . . .	164
11.2.3	dBFT . . . . .	164
11.2.4	Analysis of Dynamic Replicated Systems . . . . .	165



<b>A</b>	<b>The MS Modules</b>	<b>167</b>
<b>B</b>	<b>I/O Automata Code for dBQS</b>	<b>169</b>
<b>C</b>	<b>Proof of Correctness of the dBQS Algorithms</b>	<b>181</b>
<b>D</b>	<b>Multi-Party Secure Coin Tossing Scheme</b>	<b>187</b>
D.1	Algorithm Description . . . . .	187
D.2	Soundness Analysis . . . . .	188



# List of Figures

2-1	Example of a system organization based on the use of consistent hashing. Client $c_1$ is accessing an item with id $x$ , which is stored at the $k = 4$ servers whose ids follow $x$ in the circular id space. Client $c_2$ also accesses a second item with id $y$ , which is stored at 4 different servers.	32
2-2	Example of a problem with maintaining consistent data with a dynamic system membership. In this figure, we show a series of events that leads to old and new replica groups that only overlap in a single (potentially faulty) server.	34
2-3	Second example of a problem with maintaining consistent data with a dynamic system membership. In this example, the same sequence of membership events occur, and a long time after the membership changes, another client, $c_3$ , tries to access an old replica group that potentially contains more than the allowed number of faulty nodes.	36
3-1	In the first implementation the membership service functionality is implemented by a centralized, fault-free node.	40
3-2	In the second implementation the membership service functionality is implemented by a separate set of BFT [20] replicas.	50
3-3	In the final implementation, the membership service functionality is superimposed on the servers running in the system.	54
3-4	Different possibilities for the need to transfer the probe database across epochs. In case (a) there is no need to transfer the database, but in case (b) the database needs to be transferred.	58

4-1	The correctness conditions for servers are expressed in terms of a window of vulnerability, which is the time interval depicted in this figure.	72
6-1	BFT algorithm in normal case operation . . . . .	92
6-2	BFT view change algorithm . . . . .	94
6-3	dBFT system architecture. Clients run the client application on top of a dBFT client proxy that implements the client-side protocols. Servers run a dBFT wrapper that handles incoming messages, and a matrix of BFT-instances where rows represent different epochs, and columns represent different replica groups that the server is part of in the same epoch. Clients and servers are connected by an unreliable network. . .	96
8-1	Log-log plot of the average data maintenance bandwidth (in kilobits per second), as a function of the average membership lifetime, for three different per-node storage contributions: 1, 10, and 50 gigabytes. . .	110
8-2	Distinction between sessions and lifetimes. . . . .	112
8-3	Membership timeout, $\tau$ . . . . .	113
8-4	Membership dynamics as a function of the membership timeout ( $\tau$ ). . . . .	116
8-5	Average node availability as a function of the membership timeout ( $\tau$ ). . . . .	117
8-6	Required replication factor for four nines of per-object availability, as a function of the membership timeout ( $\tau$ ). . . . .	118
8-7	Average bandwidth required for redundancy maintenance as a function of the membership timeout ( $\tau$ ). This assumes that 10,000 nodes are cooperatively storing $10TB$ of <i>unique</i> data, and replication is used for data redundancy. . . . .	119
8-8	Required stretch (coding redundancy factor) for four nines of per-object availability, as a function of the membership timeout ( $\tau$ ). . . . .	122
8-9	Average bandwidth required for redundancy maintenance as a function of the membership timeout ( $\tau$ ). This assumes that 10,000 nodes are cooperatively storing $10TB$ of <i>unique</i> data, and coding is used for data redundancy. . . . .	123

8-10	Ratio between required replication and required stretch factors as a function of the server availability and for three different per-object availability levels. We used $b = 7$ in equation 8.4, since this is the value used in the Chord implementation [26]. . . . .	124
8-11	Instantaneous availability in the Farsite trace, for three distinct values of the membership timeout ( $\tau$ ). . . . .	125
9-1	Variation of the ping throughput as an MS replica tries to send more probes per sleep cycle. This figure shows this variation for three different levels of activity of a dBQS server co-located with the MS replica.	132
9-2	Feasibility and unfeasibility regions for different inter-ping arrival rates and system sizes. The lines above divide the deployment planes in two parts: above and to the left of the lines, the system is small enough and the probes are infrequent enough that the probing scheme is feasible with less than the threshold bandwidth. On the other side of the line, the system is so large and the probes are so frequent that the bandwidth usage surpasses the threshold bandwidth. Figure (a) assumes we use 1.7 Mbps for probes, and Figure (b) assumes we use 170 kbps. . . . .	133
9-3	Feasibility of client leases in the client population / lease duration space in three cases: (a) without aggregation, (b) aggregating 10 leases at a time, and (c) aggregating 100 leases at a time . . . . .	137
9-4	Histogram of the local time to reconfigure the system. . . . .	138
9-5	Fetch throughput, while varying the ping rate. . . . .	146



# List of Tables

3.1	System parameters for the membership service. . . . .	39
5.1	dBQS application interface. . . . .	75
8.1	Evolution of the relative growth of disk space and bandwidth. . . . .	127
9.1	Required inter-probe interval in hours for a 0.1% false positive eviction rate for different traces and different values of the membership timeout.	136
9.2	Local area network experiments. Average per object store and fetch time for different systems. . . . .	140
9.3	Average round-trip latencies between the client at CMU and different server machines. This reports the average of 20 ICMP pings, the standard deviations were under 0.3 ms. . . . .	142
9.4	The performance of read and write operations under different reconfiguration scenarios. The data reflects the average of five trials with a standard deviation of less than 2 ms. All values are in milliseconds ( <i>ms</i> )	143
9.5	Andrew100: elapsed time in seconds . . . . .	147
9.6	Time to reconfigure. These are the measured times between the client receiving a reply with the new configuration and the client executing an operation in the new configuration, for different phases of the Andrew 100 benchmark when the reconfiguration occurs. Each entry is the average of three reconfiguration times. The standard deviation was under 7% of the average. . . . .	148





# Chapter 1

## Introduction

We are increasingly dependent on Internet services provided by computer systems. These services provide important functionality and store critical state that should be made readily available to their users.

As our reliance on the Internet grows, these services become more and more attractive targets for attacks. The statistics published by the CERT research and development center for Internet security confirm that the number of attacks against Internet-connected systems is increasing: the number of reported incidents has almost doubled every year since 1997 [2].

The goal of this thesis is to develop techniques that allow Internet services to function correctly despite failures, whether they are caused by attacks, or any other causes like hardware failures, power outages, software errors, or human configuration errors. By functioning correctly we mean that services should (1) provide correct semantics, (2) be highly-available, or in other words provide continuous service, and (3) store the service state reliably, never losing any of that state.

Satisfying these requirements will require the use of replication. The basic idea is as follows. Instead of using a single server to implement a service, these techniques replicate the server and use an algorithm to coordinate the replicas. The algorithm provides the abstraction of a single service to the users, but the replicated service continues to operate correctly even when some of the replicas fail, since other replicas are still available to do the requested processing. Therefore, the system is highly

available provided no more than some number of replicas fail at the same time (the exact number of failures that can be tolerated depends on the replication protocol in use). Replication also allows data to be stored reliably since even if data is lost at some replicas it can be recovered from the others.

There has been a great deal of previous work on replication techniques. Most earlier research has focused on techniques that tolerate benign faults (e.g., [5, 36, 55, 58, 71]): these techniques assume components fail by stopping or by omitting some steps and may not provide correct service if a faulty component violates this assumption. This assumption can be problematic as malicious attacks, operator mistakes, and software errors can cause faulty nodes to exhibit arbitrary behavior. Techniques that tolerate Byzantine faults [56, 74] address this problem since they make no assumptions about the behavior of faulty components.

This thesis builds on previous work on replication, in particular, replication techniques that tolerate Byzantine faults. However, we extend this work to meet the needs of Internet services. Existing work on replication protocols, especially in the context of Byzantine fault tolerance, focuses on systems where a fixed set of servers form a single replica group that holds the entire service state. This assumption is incompatible with current Internet services, which have additional requirements.

The first requirement is *scalability*. Modern Internet services are growing in size: both in terms of the number of users that access the service and the amount of state stored by the service. For example, the Google search engine indexes 8,058,044,651 web pages (as of November 24, 2004), and handles 200 million searches per day [3]. To handle this kind of load, and to store this amount of state, Internet services must be scalable: their design should be able to accommodate a large number of servers (e.g., the Google search engine uses a cluster of more than 10,000 servers [3]), and should enable spreading the load of the system across the different servers.

The second requirement is that Internet services must deal with changes to the *system membership* (the set of servers that comprise the system at any given moment): machines will break or be decommissioned, and must be removed from the system and their responsibilities assigned to non-failed nodes, to maintain reliability and

availability of the state those nodes used to store. Also new machines must be added to the system as the need for increased storage or throughput dictates. Experience with deployed large-scale systems confirms that a dynamic system membership is a fact of life. For example, the description of the Google file system [35] (a storage substrate for Google’s services) mentions that “some [machines] are not functional at any given time and some will not recover from their current failures” [35]. Another example comes from the study of the availability of desktop PCs at Microsoft Corporation [15]. This study sent periodic ping messages to a set of machines every hour during a time interval of five weeks. The authors found that some machines would become unresponsive from a certain point in time until the end of the trace. By assuming these machines were decommissioned (or, at least, had their operating system reinstalled, which is equivalent in terms of information loss), they extrapolated the expected machine lifetime from this attrition rate, and concluded that the machines in that study had an expected lifetime of 290 days. This is equivalent to saying that, in a system of 10,000 nodes, the system will have to deal with an average of 34.5 membership changes per day.

This thesis develops new techniques that address these new challenges in the context of Byzantine-fault-tolerant systems.

## 1.1 Contributions

Our work provides a foundation for building large-scale, replicated services that handle membership changes.

We focus on services that run on a large set of servers (e.g., tens or hundreds of thousands), and are accessed by an even larger client population. The service stores a large service state, so large that it would not be feasible to store the entire state on a single machine. Instead, the service state is partitioned. Each server stores a subset of the service state, and each subset of the service state must be replicated on a number of servers for reliability and availability.

The system membership is changing throughout the lifetime of the system. These

changes must be detected automatically, and the service must adjust itself to the changes by shifting parts of the service state, to better distribute the load or to maintain replication levels for that state.

The thesis makes three main research contributions.

### 1.1.1 Membership Service

The first contribution is to solve one fundamental problem that these services face, namely keeping track of a changing system membership. This thesis presents a *membership service* that solves this problem. It is defined in a generic way so that it can be used by different services.

The membership service provides clients and servers with a globally consistent view of the set of currently available servers. The important benefit of our approach is that it allows clients and servers to agree on the current system membership, and this allows them to make consistent local decisions about which servers are currently responsible for which parts of the service state.

The membership service determines the current system membership automatically, or with minimal human intervention. This is important to avoid human configuration errors, which have been shown to be a major cause of disruption in computer systems. A recent study [72] pointed out that operator error is the largest cause of failure in many Internet services. The study explains that, in general, operator errors arose when operators were making changes to the system, e.g., scaling or replacing hardware.

Therefore the membership service is designed to monitor server availability, and automatically evict unavailable nodes from the system membership. When such membership changes occur, the membership service provides a well-defined transition from one system membership to the next, and it disseminates information about the transition to all members of the system efficiently.

An important challenge in the design of the membership service is that membership changes are themselves a threat to the correctness of the system. The membership service must both defend itself against attack and produce system memberships

whose content cannot be influenced by the attacker; otherwise an attacker might, for instance, populate the system with servers under its control, or arrange for a system membership that puts responsibility for a key part of the service in the hands of a few corrupt servers.

### 1.1.2 Replicated Services

The membership service is a useful tool for achieving our goal of building large-scale, replicated Internet services that support a dynamic system membership, but it is not enough. We still need to design services that incorporate the notion of the current system membership, allow for changes in that system membership, and adapt to these changes automatically.

The second main contribution of the thesis is to provide a generic methodology to transform a service that assumes a static membership into one that uses the output from the membership service and adjusts to membership changes.

The main challenge is to ensure that the system behaves the same with membership changes as it would with a static membership. We ensure this with a simple set of rules and generic principles that can be applied to different replication algorithms.

We also applied the methodology to two concrete examples.

We present dBQS (which is an acronym for dynamic Byzantine quorum system), a service that provides clients with read/write access to a repository of data objects. Each object is replicated at a subset of the servers and accessed with protocols based on Byzantine-fault-tolerant quorum replication [65]. As the membership changes, the subset of the servers where a particular object is replicated will also change. The dBQS service provides strong semantics (atomic access to data objects) despite Byzantine faults and membership changes.

The second example service is dBFT. This service is based on the BFT state machine replication service [20]. With state machine replication [53, 89] it is possible to build arbitrary deterministic services like file systems. However, Byzantine-fault-tolerant state machine replication algorithms were designed with the assumption of a static replica set. dBFT allows many replica groups to coexist in the system.

Furthermore, dBFT also allows replica groups to change as the system membership changes, and the replicated services maintain correct semantics despite Byzantine faults and membership changes.

### 1.1.3 Analysis of the Cost of State Maintenance

Up to this point we have simply taken for granted that it is possible for the system to adjust itself to membership changes, independently of the speed of change in the system. But we need to ask the question of how dynamic can the system be and still behave correctly? What are the negative consequences of a system that is too dynamic? Our last contribution is an analysis to answer these questions.

The resulting analysis points out an important obstacle to the deployment of systems with the characteristics that guide our design: services that maintain a highly-available replicated state, and that withstand changes in the system membership.

The basic problem that these services face is as follows. As the system membership changes, the responsibility for storing different parts of the service state shifts, and when this happens it is necessary to transfer state among the system members. This state transfer is an inevitable consequence of handling a dynamic system membership, since otherwise the replication levels for the service state would become low, and ultimately the data would be lost. The bandwidth generated by this data movement can be large, especially if the service state held by each node is large, or if the system dynamics are high. Given that nodes have a limited bandwidth to deal with this traffic, this can be an obstacle to the deployment of the system.

To evaluate how dynamic the system membership can be, we developed a model for the cost of state maintenance in dynamic replicated services, and we use measured values from real-world traces to determine possible values for the parameters of the model. We conclude that certain deployments (e.g., a volunteer-based deployment with the membership characteristics of existing volunteer-based applications) are contrary to the goals of large-scale reliable services.

We examined some bandwidth optimization strategies like using erasure codes instead of replication, and find that their gains are dependent on the deployment of

the service. For stable deployments, their gains are limited. For unstable, dynamic deployments, there are some bandwidth savings, but not enough to make the system practical: the cost of state transfer is still too high.

## 1.2 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 presents the system model and assumptions. Chapter 3 presents the membership service. Chapter 4 presents a generic methodology for transforming replicated services that work in static systems to support a dynamic membership, and we present two example replicated services in Chapter 5 (dBQS) and Chapter 6 (dBFT). Chapter 7 discusses implementation techniques for the membership service and the two example replicated services. Chapter 8 presents our analysis for the cost of membership dynamics. Chapter 9 evaluates the performance of our systems. Chapter 10 discusses related work. Chapter 11 concludes with a summary and directions for future work. Appendix A presents our programming modules that support the methodology. Appendix B presents a formal description of the algorithm that is implemented in dBQS using I/O automata [64]. Appendix C presents a correctness proof for this algorithm. Appendix D presents a secure multi-party coin tossing algorithm we developed for our membership service.





# Chapter 2

## System Model and Assumptions

This chapter presents the system model and the assumptions that underlie the design of our services.

### 2.1 Assumptions

We assume an asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, delay them, duplicate them, corrupt them, deliver them out of order, and there are no known bounds on message delays or on the time to execute operations.

We assume that nodes can use unforgeable digital signatures to authenticate communication (we denote a message  $m$  signed by node  $n$  as  $\langle m \rangle_{\sigma_n}$  and no other node can produce the signed message unless it is replaying a previous message). This assumption is probabilistic but there exist signature schemes (e.g., RSA [83]) for which it is believed to hold with very high probability. Therefore, we assume it holds with probability one in the rest of the paper.

We use a Byzantine failure model: faulty nodes may behave arbitrarily, except that they cannot forge signatures. Of course this also allows us to tolerate benign failures, since they are a special case of Byzantine failures. (In reality, it is more likely that most nodes will fail by crashing, e.g., due to machine breaking or being decommissioned, but we do not need to assume this.)

To avoid replay attacks we tag certain messages with random nonces that are signed in the replies. We assume that when clients pick nonces, with probability one they will not choose a repeated nonce.

These assumptions suffice to ensure *safety* of all the components presented in this thesis: They ensure that when a client request is executed, the execution of the request will not deviate from the specified behavior for the system. To ensure *liveness* (i.e., that client requests are eventually executed) we need to make additional assumptions about the timing of the system. For example, we can assume eventual time bounds [60] where, after an unknown point in the execution of the system, messages get delivered within certain bounds.

## 2.2 System Model

We assume a large pool of server machines that collectively provide a service to applications that are running on a large population of client machines. The replicated service stores a large service state, so large that it would not be feasible to store the entire state on a single server. Instead, the service state is partitioned (each server stores a subset of the service state). But this state still needs to be replicated on a number of servers for availability and reliability. Therefore each subset of the service state is stored at multiple servers, and it is accessed using replication protocols that ensure correct semantics despite failures. We say a server is *responsible* for that part of the state assigned to it.

On the client side there is a “proxy” for the replicated service that is responsible for the client side of the replication protocols. Applications interact with this proxy via a narrow RPC-like interface, where applications issue requests to the replicated service via the proxy, and after the request is executed the proxy will issue a reply. A client request affects only a small part of the entire service state. The client proxy must determine which servers are responsible for the relevant subset of the service state and contact them to execute the client request.

The replicated service is built using the aid of a membership service. As explained,

this is a service that provides clients and servers with consistent views of the *system membership*, i.e., the set of currently available servers.

Each server has a unique identifier (we call this the *node id*). This identifier is assigned to it by the membership service, and is used by the membership service to describe the system membership: servers are identified in the system membership using the node ids. The identifier space is very large (e.g., 160 bits) so that even very large systems can be accommodated. We view the identifiers as residing in a circular id space and we take advantage of this view to assign responsibility for work in some of our membership service protocols: we identify a task to be done by a point in the id space, and we assign the responsibility for that task to the first server in the current system membership whose identifier follows that point in the id space. We select node ids in a way that ensures they are distributed uniformly around the id space and therefore different servers will have similar probabilities of being required to perform these tasks.

As mentioned, replicated services partition the service state into smaller units; we refer to them as *items*. Items can have different granularities, e.g., an item might correspond to a file page, a file, or an entire file system subtree. The replicated service needs to assign the responsibility for items to groups of servers, which are called an item's *replica group*. The size of these groups depends on the replication protocol in use. For instance, the BFT state machine replication algorithm [20] uses groups of  $k = 3f + 1$  replicas, and works correctly provided each replica group contains no more than  $f$  faulty replicas.

Replicated services can choose to assign responsibility in various ways. For example, the service could use a directory to store a mapping, and then look up items in this directory to determine the responsible servers.

An alternative is to use a deterministic function of the system membership to assign responsibility for items. Using such a mapping simplifies the replicated service (since it does not require the use of a directory). An example of such a mapping (used in our implementation) is the use of successors in consistent hashing [48]. In this scheme, both items and servers have  $m$ -bit identifiers, which are ordered in

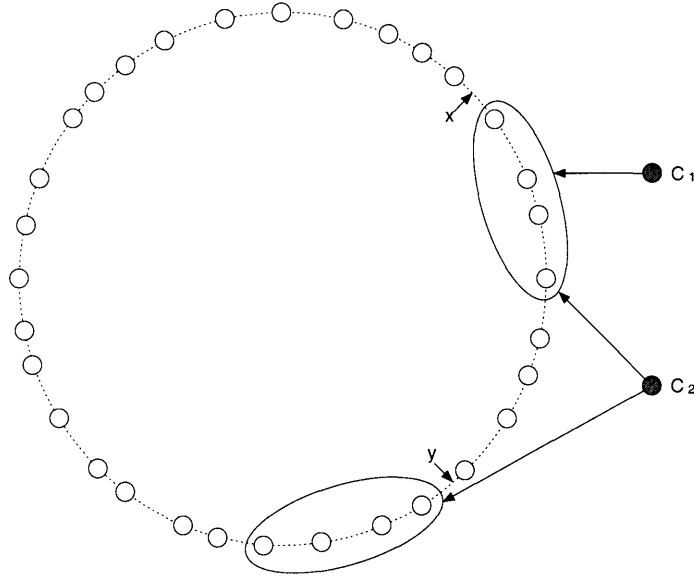


Figure 2-1: Example of a system organization based on the use of consistent hashing. Client  $c_1$  is accessing an item with id  $x$ , which is stored at the  $k = 4$  servers whose ids follow  $x$  in the circular id space. Client  $c_2$  also accesses a second item with id  $y$ , which is stored at 4 different servers.

an identifier circle modulo  $2^m$ , and the  $k$  replicas responsible for a data item with identifier  $i$  are the first  $k$  servers in the current system membership whose identifiers follow  $i$  in the identifier space (called the *successors* of identifier  $i$ ). Since the identifier space is large, we will not run out of identifiers even if the service stores many items. Furthermore, if the identifiers for servers and items are spread uniformly, then we get a nice load-balancing property that each server is responsible for roughly the same number of items [48].

Figure 2-1 depicts an example of the utilization of a system based on successors and consistent hashing. This assumes that each item is stored at a group of 4 servers (this would be the case if we used the BFT algorithm with  $f = 1$ ). The figure shows two clients,  $c_1$  and  $c_2$ , executing operations on the service. Client  $c_1$  accesses object with identifier  $x$ . The replica group responsible for  $x$  are the 4 servers that follow  $x$  in the circular identifier space. Client  $c_2$  accesses two objects, with identifiers  $x$  and  $y$ . Item  $y$  is stored at 4 other servers.

With this way to distribute the responsibility for items, it is important that node

ids are assigned in a way that is not related to physical location or other server characteristics that could lead to correlated failures. This avoids a situation where all successors for an item fail simultaneously, causing the item to be lost or unavailable. Our way of assigning node ids satisfies this constraint.

### 2.2.1 Dynamic System Membership

At any moment, a certain set of servers makes up the current *system membership*; these are the servers currently responsible for storing the service state. The system membership changes during the execution of the system. This can happen if a failure of a server is detected by the membership service, which decides to evict the server from the system membership (a mechanism we will explain in Chapter 3), or if a new server is added or explicitly removed from the system.

As the membership changes, the set of servers responsible for a particular portion of the service state also changes. For example, if some server has left the system, other servers will need to assume responsibility for the state that that server used to store. A server newly responsible for some part of the state will need to obtain a copy of that state from the servers formerly responsible for it by performing a *state transfer*.

For the service to perform correctly, we need a way for clients and servers to agree on the current system membership, since responsibility for items will depend on this, no matter how the service assigns responsibility. It is important to have different nodes agreeing on how the responsibility is assigned in the system, since replication protocols rely on certain intersection properties between the sets of replicas that were accessed by different operations to work correctly.

Figure 2-2 shows what happens when there is no agreement on the system membership. This figure shows a client,  $c_1$ , executing an operation that accesses an item with identifier  $i$  (step **a**). Then two of the servers where the operations were executed crash, and they are removed from the system membership (step **b**). After that, three new servers join the system membership, with identifiers near the neighborhood of  $i$  (step **c**). After this point, a second client,  $c_2$ , that is informed of these membership

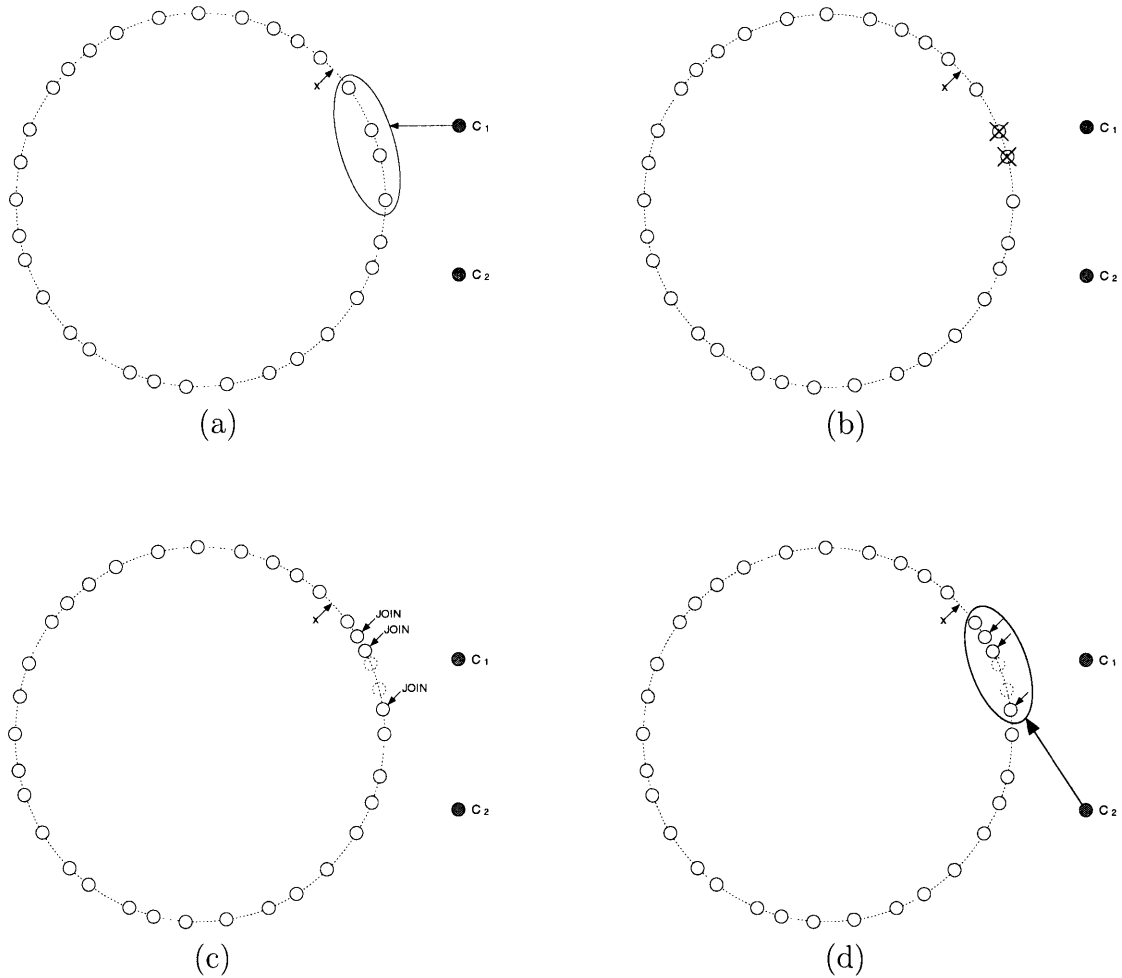


Figure 2-2: Example of a problem with maintaining consistent data with a dynamic system membership. In this figure, we show a series of events that leads to old and new replica groups that only overlap in a single (potentially faulty) server.

changes tries to access the same item  $i$  and contacts the current set of replicas for item  $i$  (step **d**). The problem is that the current set of replicas only intersects the set accessed by  $c_1$  in a single replica. This intersection is not enough to ensure that traditional algorithms work (e.g., if that replica is faulty, it can neglect to inform  $c_2$  of the operation executed by  $c_1$ ). To ensure that  $c_2$  sees the changes from the previous operation executed by  $c_1$ , we must modify replication algorithms to perform state transfer between new and old replica groups, and the second operation (by  $c_2$ ) must not be executed until after the old state has been copied from the old replicas of item  $i$ .

This scenario also illustrates a problem if  $c_1$  is not immediately informed of the membership changes, and, after  $c_2$  executes its operation, tries to access the item in its old replica group. It is required, in this case, that some of the old replicas inform  $c_1$  of the new replica group and redirect  $c_1$  to that group.

A second problem is shown in Figure 2-3. Suppose that a long time after the same sequence of events, a very slow client,  $c_3$ , that did not hear about the membership changes, tries to access item  $i$  by contacting the initial replica group for the item. In the previous example, we mentioned that the old replicas could notify the slow client of the membership changes and redirect the client to the new replicas. However, we cannot rely on this forever, since it requires that old replica groups contain no more than the allowed number of faulty nodes for a long time. If we do not impose such strict conditions, old replicas can behave arbitrarily badly in the above situation, e.g., respond that the item does not exist, or with an old state.

Note that the two problems require different solutions. For the first problem, we need only worry about what correct replicas do: they must not respond to requests about objects they are not responsible for, and, for objects they are responsible for, they must respond only after they are certain to hold the correct state.

For the second problem, though, we need to worry about very old replica groups where the number of faulty replicas exceeds the bound of the replication protocol (e.g., in the BFT state machine replication protocol, a group of  $3f + 1$  replicas with more than  $f$  faulty nodes). Here we need a way to prevent clients from using a stale system membership.

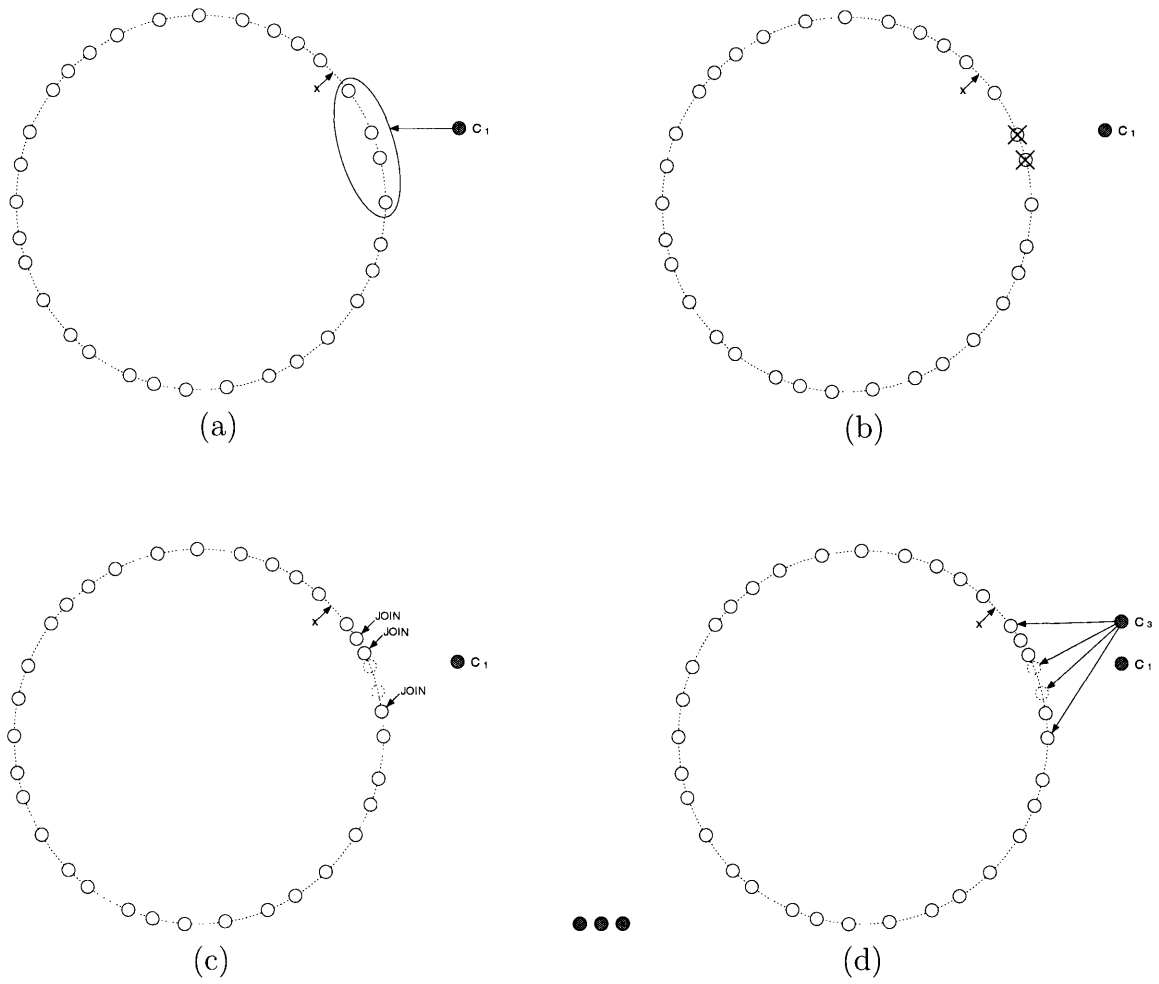


Figure 2-3: Second example of a problem with maintaining consistent data with a dynamic system membership. In this example, the same sequence of membership events occur, and a long time after the membership changes, another client,  $c_3$ , tries to access an old replica group that potentially contains more than the allowed number of faulty nodes.



# Chapter 3

## The Membership Service

This chapter describes the membership service. This service provides a trusted source of membership information, and it is designed in a way that scales to very large numbers of servers (e.g., tens or hundreds of thousands) and even larger numbers of clients.

This service is designed in a generic way that can be used by many different applications. One important class of applications that use it is replicated services with the characteristics we presented in Chapter 2: they store a large service state and make it available to a large client population. The service state is partitioned into a series of items, and each item is replicated at a subset of  $k$  servers that form that item's replica group.

In the presentation in this chapter, we will assume, without loss of generality, that the application that is using the membership service is a replicated service with these characteristics. We will refer to it as the “replicated service”.

The remainder of this chapter is organized as follows. Section 3.1 presents an overview of the membership service, and the main choices that underlie its design. Sections 3.2 to 3.4 present the design of the MS through a series of refinements. Section 3.5 presents of a discussion of some of the design choices.

### 3.1 Membership Service Overview

The membership service (MS) provides applications like replicated services with a sequence of globally consistent views of the current system membership, i.e., the set of servers that are currently available for the replicated service to run on. This set is determined based on requests to add or permanently remove servers from the system that is managed by the MS, and also on reachability information measured by the MS. Servers that are unreachable will initially be marked as inactive, and, after a long period of inactivity, will be automatically removed from the system membership.

The MS is responsible for producing descriptions of the current system membership, called *configurations*. Such a description includes the set of system members and an indication of whether they are active or inactive. The MS is also responsible for disseminating the system configuration to all the servers in the system. To allow the system configuration to be exchanged among system members without being forged, the MS authenticates it using a signature that can be verified with a well-known public key.

A key part of our design is that the MS only produces new configurations occasionally, and not after every membership change event. The system moves in a succession of time intervals called *epochs*, and we batch all configuration changes at the end of the epoch. Epochs are advantageous for several reasons. They simplify the design of the replicated service, since it can assume that the system membership will be stable most of the time, and the replicated service can be optimized for periods of stability. Also periodic reconfiguration reduces some costs associated with propagating membership changes (like authenticating configurations or transmitting them). Each epoch has a sequential *epoch number* associated with it that is included in the system configuration.

Our design is flexible about what triggers the end of an epoch. Epochs can have a fixed duration; or they can end after a threshold of membership changes in the system is exceeded. We can also use a hybrid scheme where epochs end upon the first of the two conditions: exceeding a threshold of membership changes, or exceeding the

Parameter	Meaning
$T_{epoch}$	Duration of an epoch
$l_{add-cert}$	Number of epochs for which add certificates are valid
$n_{inactive}$	Number of failed probes to become inactive
$T_{probe}$	Interval between probes
$n_{remove}$	Number of inactive epochs before removing
$T_{lease}$	Lease duration
$f_{MS}$	Maximum number of faults in MS replica group
$n_{start-inactive}$	Number of failed probes to initiate eviction

Table 3.1: System parameters for the membership service.

maximum duration.

Our implementation uses a fixed epoch duration which is a parameter,  $T_{epoch}$ , set by the system administrator. (For reference, we list all the system parameters of the membership service in Table 3.1.) This epoch duration must be chosen based on the characteristics of the servers in the deployment of the system. For stable environments, we can use relatively long epochs (e.g., an hour). This is because we can set the replication factors used by the service in a way that the remaining servers ensure the service availability. Chapter 8 elaborates on how to set the replication factors depending on how long we wait to remove unavailable members from the system.

The remainder of this chapter describes three implementations of the MS. In the first implementation (Section 3.2) the MS is implemented by a single, fault-free node. This allows us to show how the basic functionality can be implemented, but has very stringent correctness criteria. In the second implementation (Section 3.3) the MS is implemented by a Byzantine-fault-tolerant replication group, which adds complexity to the implementation but allows us to tolerate more failures in the nodes that implement the MS. In our final refinement (Section 3.4) we superimpose the replica group on the server nodes that implement the replicated service, and we allow the servers that comprise the replica group to change.

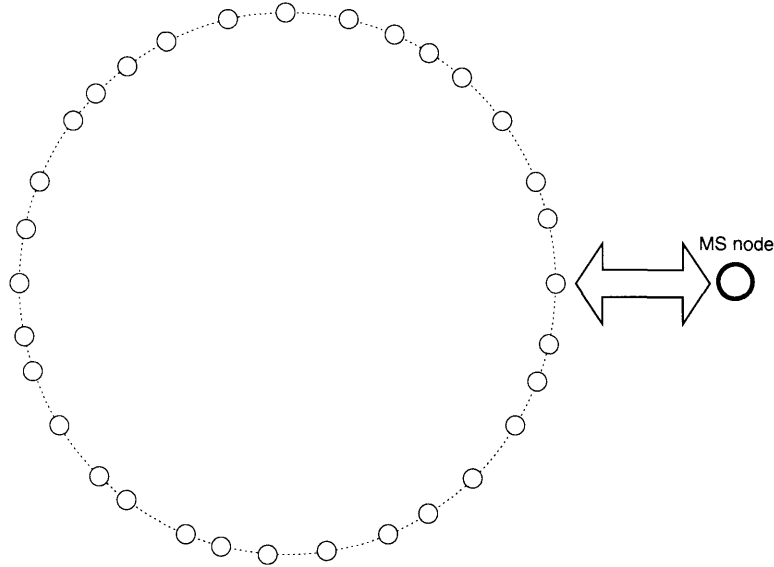


Figure 3-1: In the first implementation the membership service functionality is implemented by a centralized, fault-free node.

## 3.2 Centralized Membership Service

Here we assume the membership service is implemented by a separate, fault-free node that we will call the MS node. This node is the sole entity responsible for determining membership changes and producing configurations for the system. These configurations are sent to the system nodes, and can be exchanged among them. This system organization is depicted in Figure 3-1. The figure shows a service formed by several servers with identifiers in a circular space, and a separate MS node that implements the membership service.

This implementation has very strict correctness conditions: we assume that the MS node never fails. We will relax this assumption in subsequent implementations of the MS.

### 3.2.1 Membership Changes

The MS needs to respond to requests to add and remove nodes, and it also needs to detect unreachable nodes and remove them automatically.

## Explicit Addition and Removal

Our design requires a form of admission control to join the system. We excluded an open membership system since these are vulnerable to a Sybil attack [29] where an adversary floods the system with malicious nodes. In such a setting it would be nearly impossible to ensure that the correctness conditions of replicated services that use the MS are met.

Therefore we require that nodes can only be added by a trusted authority, and that the trusted authority must prevent attacker from easily adding a large number of nodes to the system. This can be done, for instance, by ensuring a relationship between nodes in the system and real-world entities.

We employ a simple trust model based on public key cryptography: we assume that the trusted authority has a well-known public key that can be used to validate certificates it issues. To reduce the risk of exposing certificate signing keys, the trusted authority should produce certificates offline; thus the machine used to produce these certificates should not be involved in the regular operation of the system.

The certificate for adding nodes to the system contains the information needed for clients and servers that are running the replicated service to establish authenticated communication with the new node: the network address and port number of the new node, and its public key. The certificate also includes a random number that is generated by the trusted authority, which will be useful for assigning node ids.

To be added to the system a node must first obtain the add certificate from the trusted authority. Then the incoming node forwards this certificate to the MS node.

When the MS node receives a request to ADD a member it must verify the authenticity of the respective certificate. If this is authentic, the information about the node is recorded in a list of nodes that must be added to the next configuration. The MS node also generates a node id for the incoming node, which is included in the system configuration. Node ids are generated as a deterministic function of the random number included in the add certificate. This function should generate node ids that are uniformly distributed in the circular id space. The simplest scheme is to

use the random number as the node id, but we also allow more elaborate schemes. For instance, we can obtain better load-balancing by generating a number of ids by hashing the random number and different salt values, and choosing the id that more evenly spaces nodes in the id space given the current system membership [49]. Our implementation just uses the random number as the node id.

The trusted authority can also permanently REVOKE membership (e.g., of nodes that are known to be compromised) by issuing revocation certificates. This certificate needs to identify the node to be removed by its public key. Such a certificate is also forwarded to the MS node after it is produced. The MS node also validates the certificate and records the information about the node that will be removed in the next configuration.

We need to prevent replay of add requests; otherwise a node that was previously removed from the system could be re-added by a malicious party. We do this by putting an interval of epoch numbers in the add certificate. The length of the interval (the number of epochs during which the certificate is valid) is a system parameter,  $l_{add-cert}$ . An add request is only good while the epoch number of the current configuration falls within the interval in the add certificate; otherwise it is rejected. This means we need to remember revocations for  $l_{add-cert}$  epochs. If we set  $l_{add-cert}$  to 1, then we do not need to remember revocations in future epochs.

## Automatic Detection of Unreachable Nodes

Besides processing explicit add and revoke requests, the MS must also automatically detect unreachable nodes and mark them as INACTIVE. It does this by probing system nodes periodically, round-robin. The time interval between two consecutive probes to a system member is a system parameter,  $T_{probe}$ . The probes are usually unauthenticated ping messages, which we expect to be sufficient to detect most unreachable nodes. Infrequently, probes contain nonces that must be signed in the reply, to avoid an attack that spoofs ping replies to maintain unavailable nodes in the system. Signed pings are used sparingly since they require additional processing on the MS node to verify signatures. However, once a node fails to reply to a signed ping, all subsequent

pings to that node must request signatures (until a correctly signed response arrives).

Probe results are inserted into a *probe database* that records, for each system member that is currently unreachable (or, in other words, that has failed to reply to the last probe that was sent to it) how many consecutive probes that member has failed to respond.

If a node fails to reply to a threshold of consecutive probes, it is declared to be inactive and will be added to a list of inactive nodes. This threshold is a system parameter,  $n_{inactive}$ .

Marking nodes as inactive must be done slowly, which means that the probing period  $T_{probe}$  and the threshold  $n_{inactive}$  should be set by the system administrator such that the time to mark a node inactive,  $T_{probe} \cdot n_{inactive}$ , is relatively high. For applications like the ones we envision being the target users of our MS (i.e., replicated services that store large amounts of service state), it is important to have a *delayed response to failure* for several reasons: It avoids unnecessary data movement due to temporary disconnections, offers additional protection against denial of service attacks (assuming we wait for longer than the duration of such attacks), and avoids thrashing, where in trying to recover from a host failure the system overstresses the network, which itself may be mistaken for other host failures, causing a positive feedback cycle. The downside of using a delayed response to failures is that the average availability of active system members will be smaller than if we had a quick response to failures. However, this can be circumvented by setting  $f$  (the redundancy parameter for the replicated systems) appropriately. This issue is addressed in more detail in Chapter 8.

However, we also envision the use of our system by other applications that want to be more aggressive about marking unreachable nodes as inactive. It would be simple to extend the design to include a parameter  $T_{fast-probe}$  that represented a shorter probe interval that would be used after the first failed probe to a system member. The system would revert to the normal probe rate after a correct reply was received from that node. Note that we cannot decrease the probe period drastically, or else we can suffer from the thrashing effects described above.

If an inactive node contacts the MS node, it will be marked as RECONNECTED.

The MS node must send a challenge to the reconnecting node that is signed in the reply to avoid a replay attack. Also nodes are not allowed to remain inactive indefinitely; instead, if a node has remained inactive for longer than a number of epochs (determined by the system parameter  $n_{remove}$ ), it is automatically removed from the system membership.

## Byzantine Fault Detection

Probes are a good mechanism to detect nodes that crashed or that are unreachable. We cannot rely on them to detect Byzantine faults, since a Byzantine-faulty node might respond correctly to them.

In general, Byzantine fault detection is a hard problem because the system can be a target of a *lying in wait attack*, where an attacker compromises more and more nodes, making them behave correctly until a sufficient number of nodes have been corrupted that launching an attack at that point would cause substantial damage to the system. Therefore we cannot rely on a node's external behavior to determine if it is compromised.

Even though there is no complete solution to the problem of detecting Byzantine faults, we point out a simple extension to the scheme above that would allow us to detect some of these faults.

This extension is based on proposals for *remote attestation* [22, 33, 1]. These proposals rely on secure hardware and a small, trusted OS kernel base to produce a certificate that enables a machine to prove to a remote system what local programs are running.

The idea of remote attestation is that nodes run tamper-resistant hardware [34]: the processor has a private key that changes in the case of an attack on the hardware. The processor contains a special register that contains a fingerprint of the software that is running in the system, and there is a special instruction the processor can execute that receives a nonce as an argument, and outputs a signature of the nonce and the fingerprint of the running code.

Most proposals for remote attestation assume short-lived sessions between clients



and servers where the attestation is only performed in the beginning of the interaction between the client and the server, e.g., to ensure that both nodes are running the most up-to-date versions of the operating system and the application software.

We would like to use remote attestation in a slightly different way, as a means to automatically remove server nodes that have been compromised, and divert the clients to other, correct servers.

To do this we augment our probing scheme with remote attestation. This is done by piggybacking the output of the software attestation instruction on the signed probes. When the MS receives a reply to a signed probe, it checks to see if the software running on the server is the same as what is expected.

If the reply is correctly signed but the node is not running the correct software, it can be immediately added to the list of nodes to be removed from the system. If the reply does not contain a correct signature, it is ignored, and after a number of failed replies the node is eventually removed from the system.

### 3.2.2 Propagating the System Membership

The MS node must propagate information about membership changes to the servers at the end of each epoch.

Our implementation has a fixed epoch duration,  $T_{epoch}$ . The MS node starts a timer with that duration at the beginning of an epoch, and STOPS the epoch when that timer expires. Note that, as mentioned, other conditions for ending epochs are possible, e.g., exceeding a number of membership changes, or a threshold of inactive system members. It is easy to extend the current implementation to end epochs abruptly upon such conditions.

When the timer expires the MS node stops probing, and produces a certificate describing the membership and epoch number of the next configuration. This certificate is signed with the MS node's private key.

Then the MS sends a message to the other nodes describing the next configuration. This message describes the configuration change using deltas:

$\langle epoch\ number, add\ list, drop\ list, unreachable\ list, reconnected\ list, \sigma_{MS} \rangle$

where *add list* is a list of the nodes that joined the system during that epoch, and contains, for each node, its network address, port number, public key, and node id; *drop list* is the list of the node ids of nodes that were permanently removed (either because a revocation certificate was issued, or because they were inactive for over  $n_{remove}$  epochs); *unreachable list* is the list of ids of nodes that became inactive during this epoch; *reconnected list* is the list of ids of nodes that used to be inactive but reconnected during this epoch; and  $\sigma_{MS}$  is a signature certifying that the MS produced the configuration.  $\sigma_{MS}$  is not a signature over the message described above, but over the entire configuration, consisting of the following:

- The current epoch number;
- and, for each system member:
  - its node id;
  - its network address;
  - its port number;
  - its public key;
  - a flag indicating if it is active or inactive.

This list is sorted by node id.

When a node receives this message, it produces the new configuration from its old one and the delta and then checks the signature. Transmitting only deltas is important for scalability.

To avoid an expensive one-to-all broadcast, we disseminate the certificate using multicast trees (made up of servers). For increased fault-tolerance we set up  $m$  redundant trees rooted at the MS node: each system node is included in more than one tree. However, nodes can also get a certificate from any other node that already received it, and we make use of this ability in the replicated services presented in later chapters.

The multicast trees are setup as follows. The MS node chooses  $m$  random numbers in the node id space. Then it sends a level 0 message to the nodes in the next

configuration that succeed each of the  $m$  ids by  $i \cdot \sqrt[3]{N^2}$  nodes, where  $N$  is the number of nodes in that configuration, and  $i = 0, \dots, \sqrt[3]{N} - 1$ . Nodes receiving level 0 messages send level 1 messages to the nodes that succeed it by  $i \cdot \sqrt[3]{N}$ , again with  $i = 0, \dots, \sqrt[3]{N} - 1$ . Finally, nodes receiving level 1 messages send level 2 messages to their  $\sqrt[3]{N}$  immediate successors. This guarantees that each of the  $m$  trees spans all system nodes.

The number  $m$  of redundant multicast trees that are set up represents a tradeoff between the likelihood that all system members will receive the new configuration by this process, and the amount of redundant information that is transmitted.

### 3.2.3 Supplying Old Configurations

As mentioned, replicated services need to do state transfer at epoch boundaries. But to do state transfer, a node needs to know not just the configuration for the current epoch but also the previous one, since it needs to know which nodes were previously responsible for items that are now its responsibility.

However, a node might miss hearing about some epochs. When it finally learns about the new configuration, it will need to learn the missing information, both in order to compute the membership of the current epoch (since we send deltas) and to learn about previous epochs.

This implies that the system needs to remember previous configurations. We place this responsibility on the MS. Since configurations are authenticated, we can use unauthenticated requests and replies to obtain them.

For now we will assume that the MS maintains all configurations since the beginning of the system, and that each server node can obtain any configuration from the MS node. This can be a large amount of state if the system lasts for many epochs. Section 3.5.2 describes a scheme to garbage collect old configurations when they are no longer needed.

### 3.2.4 Freshness

Our approach of using epochs and certificates that describe the system membership for each epoch simplifies the design of replicated services. By augmenting the replication protocols to include epoch numbers in requests we ensure that two correct nodes that are in the same epoch agree on the system membership (hence they agree on the set of replicas for the operation in question). If the client is still behind the servers it is contacting, it can request the certificate for the new configuration from the servers, and consequently upgrade to the new epoch.

However, as mentioned in Section 2.2, this does not solve the problem entirely, in particular if clients are lagging behind by several epochs. This can happen for several reasons, e.g., if the client is slow and has not been receiving messages from other nodes, or when a client joins the system for the first time, and has just downloaded the configuration from another running node. This can lead to the problematic situation explained in Figure 2-3 where a client with a stale configuration contacts an old replica group to execute an operation, and the failure threshold in that group (i.e., the number of faulty servers in the group) has long been exceeded.

We could address freshness by having each server use a new public (and associated private) key in each configuration (as is done in [67]). In this scheme, the membership service picks new public and private keys for all replicas when replica groups change, and, for each replica, encrypts the new private key with the old public key of the replica, and sends it to that replica. Each (non-faulty) server discards its old key after receiving its new one, preventing clients from executing operations in old groups (assuming enough replicas discarded their keys). But this approach is not scalable when there is a large number of servers, both because of the communication required between the MS and servers when the membership changes, and because of the size of the configuration message. Also this approach makes it more difficult to replicate the MS (as described in Section 3.3), since faulty MS replicas would have access to critical information.

A variation that would allow the MS to be replicated would be for servers to choose

the public/private key pairs, and send the public key to the MS. But this approach has the problem that the MS could not make progress if a faulty node would not send it its public key.

Therefore we use another approach, *leases*. The mechanism works as follows.

Clients obtain leases by issuing challenges to the MS. The challenge contains a random nonce; the MS node responds by signing the random nonce and the epoch number of the current configuration. The response from the MS gives the node a lease during which it may participate in the normal system operation. The duration of the lease is a system parameter,  $T_{lease}$ . As future work, we could make the duration of the lease adaptable, so that leases would expire shortly after the end of the current epoch. In designing such a mechanism we need to be careful not to overload the system after epochs end.

A client will only accept and process a reply from a server if it holds a valid lease (our client side code for the replicated services enforces this). We must set the epoch duration and the lease duration in a way that ensures that clients will never try to contact replica groups where the failure threshold has been exceeded (i.e., they contain more than  $f$  faulty nodes). Note that leases do not prevent new epochs from being produced. They only limit the time a client will use an epoch it already knows.

Leases are required by clients for correctness, but servers can also use them as a means to speed up the discovery of new configurations. If a server is using the same configuration for a while and has not checked with the MS to ensure it is still valid, its lease will expire and this will cause the server to issue a new challenge, and find out if its configuration is still up-to-date. If not, the MS will forward the most up-to-date configuration to that server.

### 3.3 Replicated Membership Service

The current design has an obvious problem: The MS node is a single point of failure. In this section we replace the MS node with a group of  $3f_{MS} + 1$  MS replicas executing the Castro-Liskov BFT state machine replication protocol [20], We assume that these

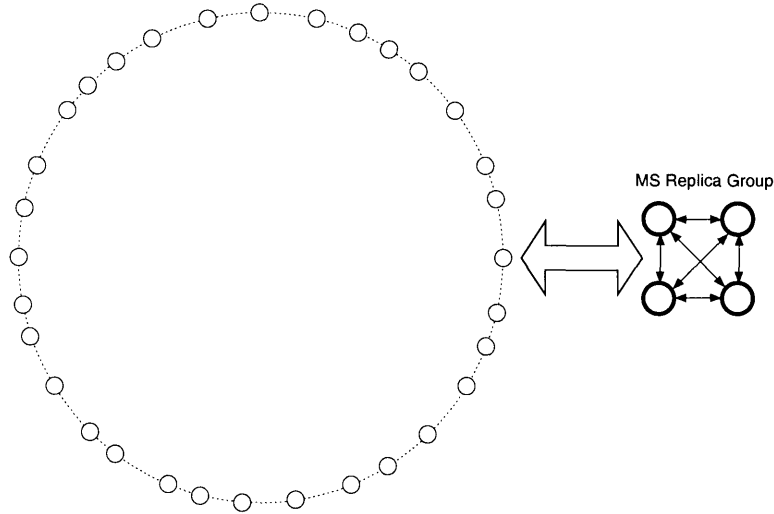


Figure 3-2: In the second implementation the membership service functionality is implemented by a separate set of BFT [20] replicas.

MS replicas are always the same machines, and that no more than  $f_{MS}$  replicas become faulty throughout the entire execution of the system. We also assume that each MS replica has its own private key, and the corresponding  $3f_{MS} + 1$  public keys are well-known. This system organization is depicted in Figure 3-2.

### 3.3.1 Membership Changes

Moving to a BFT group requires some changes in the previous design. First, the ADD and REVOKE operations must be performed as BFT state machine operations. The request can be sent to one (or more) of the MS replicas, which invokes the state machine operation on the BFT service. The state machine operation to add a system member or permanently revoke membership takes as argument the add or revoke certificate, respectively. The execution of the operation validates the certificate: it verifies that it is correctly signed, and, in the case of an add certificate, that the range of epochs in the add certificate includes the current one. For revocation operations, the BFT service verifies that the node is a current member of the system.

In the replicated MS, replicas probe system members independently, which is needed so that they can monitor one another and so that we do not need to trust a

single replica that may be lying. To mark the node inactive, an MS replica must get the other MS nodes to agree, and this is not trivial, since different nodes will have different values for how long each node has been unreachable.

MS nodes initiate the process to mark the node inactive if  $n_{start-inactive}$  consecutive pings for that node fail. (Like  $n_{inactive}$ ,  $n_{start-inactive}$  is also a system parameter set by an administrator.) In this case, that MS replica proposes the node should be marked inactive to other MS replicas. Then, it must collect signed statements from at least  $f_{MS} + 1$  MS replicas (including itself) that agree to mark that node inactive. Other MS replicas accept the proposal (and sign a statement saying so) if the last  $n_{inactive}$  pings for that node have failed (according to their own probe database), where  $n_{inactive} < n_{start-inactive}$ .  $n_{inactive}$  and  $n_{start-inactive}$  should only differ by one or two units so that nodes are detected as unreachable in a timely way. This approach ensures that the proposal will usually succeed if the node is really down.

Once a node has the necessary signatures, it causes the node to be marked as inactive by running an operation on the BFT service, for the purposes of agreement and serialization with other operations we will describe next. The INACTIVE operation that is invoked on the BFT service has two parameters: The identifier of the node being marked inactive and a vector  $\langle \sigma_1, \dots, \sigma_{f_{MS}+1} \rangle$  containing at least  $f_{MS} + 1$  signatures from MS replicas agreeing to mark the node inactive during the current epoch. The operation will fail if there are not enough signatures or if they do not verify.

The reconnect operation is performed by a replica when it hears from an inactive node. This also requires a vector of  $f_{MS} + 1$  signatures of MS replicas willing to reconnect the node during that epoch. MS replicas will produce these signatures after sending a challenge to the reconnecting node, and receiving a correctly signed reply. When the operation is executed, the MS verifies that the node is an inactive system member, and verifies that there are enough correct signatures.

If a node is marked inactive and then reconnects in the same epoch, we need to be careful not to accept the old vector of signatures that was used to mark the node

as inactive in a replay that would make the node inactive again. This could be done by adding sequence numbers in the inactivity statements, but instead we only allow nodes to be marked inactive or reconnected no more than once per epoch.

### 3.3.2 Propagating System Membership

The STOP operation also needs to be changed since it requires agreement on when to end the epoch. We use the same approach as in INACTIVE of collecting signatures agreeing to stop epoch  $e$ , but in this case each node consults its local clock to determine how long it thinks the epoch has lasted, and agrees only if it has lasted more than  $T_{epoch} - \delta_{epoch}$ , where  $\delta_{epoch}$  is a system parameter that allows for the operation to succeed despite a small drift in clock rates. Note that we do not assume synchronized clocks, nor synchronized clock rates. However, if there are more than  $f_{MS}$  non-faulty MS replicas with a slow clock rate, this might cause the STOP operation to be executed later, causing the system membership to change later than desired.

After the STOP operation is executed, all MS replicas agree on the membership in the next epoch: nodes for which REVOKE operations have been executed are removed and those for which ADD operations have been executed are added. Also INACTIVE and RECONNECT operations mark nodes as inactive or active, and we remove nodes that have been inactive for longer than  $n_{remove}$  epochs.

Then the MS replicas can produce a certificate describing the membership changes for the new epoch similar to the one described before, except that the certificate can no longer be signed by a single node, since we cannot trust any particular node. Instead, we sign the configuration with a vector of at least  $f_{MS} + 1$  signatures from MS replicas.

This certificate is disseminated using multicast trees as before, but now each MS replica sets up a single multicast tree to disseminate it. Each tree spans all system members, therefore our current implementation uses  $m = 3f_{MS} + 1$  redundant trees (in some cases, when MS nodes are faulty or slow, the number of redundant trees may be smaller). Note that we could increase the number of redundant trees by having



each node start more than one tree that spanned all members.

### 3.3.3 Leases

The challenge-response mechanism for granting leases is still implemented by the MS, but now leases require a vector of  $f_{MS} + 1$  signatures.

### 3.3.4 Correctness of the Membership Service

This implementation of the MS will provide correct service if no more than  $f_{MS}$  MS replicas fail during the entire system lifetime.

We can weaken this condition using an extension to the BFT state machine replication algorithm called *proactive recovery* [21]. Proactive recovery assumes that each replica of the BFT group (in this case, each MS replica) has a secure coprocessor that holds its private key and a watchdog timer that periodically (e.g., every 10 minutes) interrupts processing and hands control to a recovery monitor, which restarts the node. When the node is restarted, the coprocessor reinitializes the code running on the main processor from a copy on a read-only disk. The idea is that at this point the code is correct. Then the node runs a restart protocol that corrects its copy of the service state if that has been corrupted.

Proactive recovery allows the BFT service to work correctly despite an arbitrary number of faults, provided less than  $f_{MS}$  replicas fail within a small window of vulnerability (roughly equal to the restart period). However, it opens an avenue for an attack where an adversary compromises one MS node at a time and gets each of them to sign a wrong membership list for a future configuration.

In the next section, we will explain techniques that overcome this problem.

## 3.4 Reconfigurable Membership Service

The use of proactive recovery with a fixed replica group allows us to tolerate an arbitrary number of faults among these replicas, but it still does not solve the problem

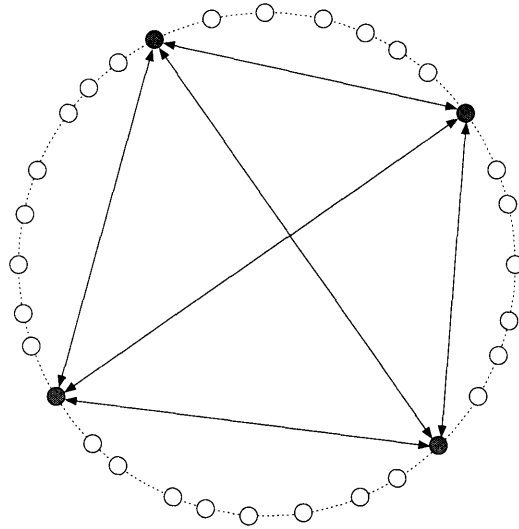


Figure 3-3: In the final implementation, the membership service functionality is superimposed on the servers running in the system.

of an MS replica failing permanently, and needing to be replaced. In this case we would like to automatically reconfigure the MS, replacing the failed MS replica with a new, non-failed node.

Our (final) refinement addresses this by changing the MS membership. We do this by periodically moving the MS to a new set of nodes, as part of transitioning between epochs. Our current implementation changes the MS replicas every epoch, but we can also change it less frequently, every  $n_{change}$  epochs.

Our design superimposes the MS on the current set of system members: we assume servers occasionally act as MS replicas, in addition to running the replicated service.

However, we could easily limit what servers are allowed to run the MS by marking servers (in the add certificate) to indicate the roles they are allowed to assume. These marks could be used to allow the MS to run only on particularly capable or well-connected servers.

Superimposing the MS on the system members ensures we have a pool of available servers on which to run the MS. Superimposition also provides good security: it allows us to move the MS periodically as a way of avoiding an attack directed at a particular set of replicas.

### 3.4.1 Choosing the New Membership Service

The MS is a particularly tempting target to attack because if more than  $f_{MS}$  of its nodes are faulty, the entire system will fail. Therefore we would like to make the attack difficult.

To achieve this goal we move the MS periodically. At the end of each epoch we choose a new set of system members who will implement the membership service for the next epoch. However, it would be bad if the attacker could predict where the MS is running in advance since this would allow it many epochs to launch an attack. Therefore we want to move the MS in an unpredictable way. Our solution is to choose the MS based on a random number. Once this number has been chosen, the MS replicas for epoch  $e + 1$  are chosen to be the nodes whose ids follow (or equal)  $h(i, r_{e+1})$ , where  $i \in \{1, \dots, 3f_{MS} + 1\}$  is the replica number,  $h$  is a random hash function (our implementation uses SHA-1), and  $r_{e+1}$  is the random number chosen for epoch  $e + 1$ .

It might seem that we could compute this random number as a hash of the configuration membership. But our system might run in an environment with very low churn, and the membership might not change for several epochs. Therefore this approach does not work and we require a more secure way of computing  $r_{e+1}$  that ensures that it is impossible to predict  $r_{e+1}$  before it is computed and that no compromised node can bias the choice of that number. For this purpose we use a multi-party secure coin tossing scheme [77].

The reason why we use  $h(i, r_{e+1})$  instead of a simpler scheme (e.g., picking the successors of  $r_{e+1}$ ) is to avoid a total match between the MS replica group and a replica group for the replicated service, since the replicated service may also pick the same simple scheme like successors. If there were a total match, that group of the replicated service might suffer performance problems due to the extra work of implementing the MS.

### 3.4.2 Discovering the MS

As mentioned, the MS must supply old configurations to slow servers that have missed some epochs and need their configurations for state transfer (and to verify the authenticity of the configuration delta).

With the reconfigurable MS this is problematic since the slow servers cannot know who the MS replicas are without computing the current configuration. To break this circular dependence we augment the current configuration with a signed list of the  $3f_{MS} + 1$  nodes that form the MS.

### 3.4.3 Signing

Another issue that arises in this refinement is how configurations are authenticated. A vector of signatures no longer works since incoming nodes do not know who the current MS replicas are; this means a group of bad nodes could pretend to be an MS.

Therefore we switch to having the configuration be validated with the MS public key. Each MS replica holds a share of the associated private key and the signature is produced using a proactive threshold signature scheme [45]. This scheme will only generate a correct signature if  $f_{MS} + 1$  replicas agree on signing a statement. When epochs change, the new replicas obtain new shares from the old replicas, allowing the next MS to sign. Non-faulty replicas discard old shares after the epoch transition completes. Different shares must be used by the new MS replicas because otherwise these shares could be learned by the attacker once more than  $f$  failures occurred in a collection of MS groups.

Another point is that we need to be careful in carrying out the protocol that moves the information about the shares to the new MS members. If this information were sent by encrypting with their public keys, then eventually an attacker could figure out the secret, by saving all messages. Then when a node previously in the MS became faulty, the attacker could get it to decrypt all messages sent to it earlier. As soon as  $f_{MS} + 1$  nodes from the same configuration have done this, the attacker would know the secret.

We solve this problem by having the new MS members choose new public/private key pairs. These are their *epoch keys*; they are distinct from the nodes' permanent keys (which never change). MS members use their epoch keys to encrypt the information they exchange in the resharing. The old MS waits to hear about at least  $2f_{MS} + 1$  new public keys from the new members before doing the resharing. For agreement on these keys, new replicas must run an operation on the old MS state machine to submit these keys. And as soon as the old MS produces the new configuration, all its honest replicas discard their epoch keys.

Epoch keys for the new MS replicas are included in the configuration. Since there is only a small number of them, this does not affect scalability.

The challenges used to obtain leases are now signed with the epoch keys. We also use epoch keys to sign the arguments to the INACTIVE, RECONNECT, and STOP state machine operations.

Using epoch keys instead of threshold signatures for these functions is advantageous since the cost of producing a vector of signatures is lower than producing a threshold signature. Therefore we only use threshold signatures to authenticate configurations.

This approach makes the configuration a bootstrap of trust in the system: it can delegate trust to the epoch keys by signing the corresponding temporary public keys. Furthermore anyone using the system can trust the configuration since we know its threshold signature must have been produced by one of the replica groups in a chain of trustworthy MSs.

### 3.4.4 State Transfer

Our MS has state: it knows about the current and previous configurations, the list of membership changes to be applied in the next configuration, and its members have their probe database. So the question is how to transfer this state when the MS moves?

The next configuration is signed by the threshold signature scheme and propagated to new MS members, and the lists of membership changes are reset when epochs

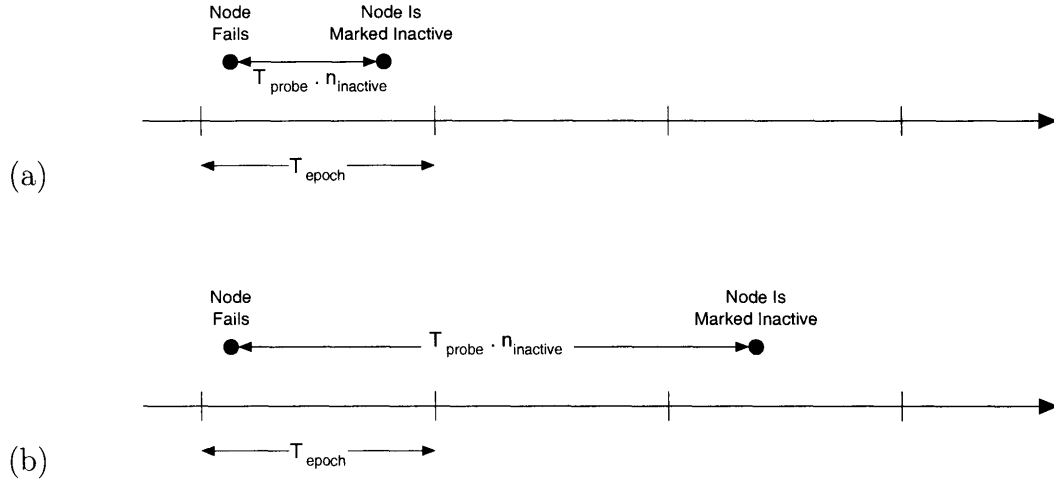


Figure 3-4: Different possibilities for the need to transfer the probe database across epochs. In case (a) there is no need to transfer the database, but in case (b) the database needs to be transferred.

change, so the current configuration information is simple to transfer.

Old configurations can be obtained directly from any non-faulty MS member directly since they are authenticated. This can represent a substantial amount of information if the system is running for many epochs. We address the issue of garbage-collecting old configurations that are no longer needed in Section 3.5.2.

The other piece of MS state that may need to be transferred is the probe database. Figure 3-4 illustrates two different cases that need to be distinguished. If the time to mark a node inactive after its unreachability started (in our implementation this is equal to  $T_{probe} \cdot n_{inactive}$ ) is smaller than the epoch length,  $T_{epoch}$ , we can discard the probe database (part (a) of the figure). The loss of ping information means that some work is lost (some pings will have to be re-sent), but that is not a problem since epochs are long enough so that unresponsive nodes will be declared inactive in the new epoch.

If the time to mark the node inactive is larger than  $T_{epoch}$  (part (b) of the figure), the probe database needs to be transferred to ensure that the process of detecting the inactivity of the node will eventually succeed. This process is complicated by the fact that this is a non-deterministic state: different MS replicas will have different values for how long nodes have been unreachable.

Our solution is to have old MS replicas invoke an operation on the BFT service that submits the values of their probe database to the old MS. (This state is relatively small since only the nodes that have failed their last contact attempt are in the probe database.) After  $2f_{MS} + 1$  old MS replicas have invoked the operation, the new MS replicas can invoke an operation on the service to read their initial values for the probe database. The latter operation will return, for each currently unreachable node, the median value for the number of consecutive failed attempts to contact that node (among the  $2f_{MS} + 1$  values that were submitted, considering that a node that was not reported is unreachable for zero attempts). This discards  $f_{MS}$  possible high values or  $f_{MS}$  possible low values from faulty replicas.

### 3.4.5 Epoch Transition Protocols

We now describe how the epoch transition protocol works, by enumerating the sequence of steps, and explaining each step in detail.

1. *Run the stop operation and submit values for the probe database.*

This protocol starts when the first MS node invokes the STOP operation on the MS. After this operation runs, all the old MS nodes that execute this operation will run an operation to submit their probe database values. Once  $2f_{MS} + 1$  MS nodes have run this operation, the service discards all the values that were submitted for the probe databases with the exception of the median values, which will be transmitted to the new MS replicas.

2. *Pick the random number.*

At this point the old MS replicas run the multi-party secure coin tossing scheme [77] to pick the random number to be included in the next configuration,  $r_{e+1}$ .

3. *Notify the new replicas to start the protocols.*

After  $r_{e+1}$  is chosen, any old MS replica can compute who the new MS replicas are. Each old MS replica will individually send a message to all the new MS replicas, requesting that they initiate their part of the protocols. These messages are retransmitted if necessary until acknowledged (or if the epoch transition concludes).

New MS replicas will wait until they receive  $f_{MS} + 1$  notifications to start their side

of the epoch transition protocols, to avoid being spoofed into starting this protocol when they should not.

*4. New replicas read the probe database and propose epoch keys.*

The first thing that new MS replicas must do is run operations on the old service to submit their epoch keys and read the probe database values that were recorded in the old service. These will be the initial values for their probe databases.

We cannot wait for all new MS replicas to submit their epoch keys since  $f_{MS}$  of them can be faulty and never do it. Instead, after the new replicas have run  $2f_{MS} + 1$  operations to submit their epoch keys, we wait for a fixed time (a parameter  $\delta_{wait}$  that can be set to only a few seconds) for the remaining new replicas to submit their keys. After that time, or after all keys are present, we move to the next steps of the protocols. The additional wait is just a performance optimization: the more epoch keys are known the better we can balance load on the new MS. However, the system works correctly with only  $2f_{MS} + 1$  epoch keys in the configuration.

*5. Perform the share refreshment protocol.*

Once we have enough epoch keys and we have waited the additional  $\delta_{wait}$ , the old replicas start the share refreshment protocol that produces new shares from the old ones. All the communication sent to the new replicas in this protocol is encrypted using the epoch keys for the new replicas.

A problem that may arise is that up to  $f_{MS}$  replicas may be correct but slow and not submit their epoch keys in time to be for these keys to be included in the configuration. As mentioned, this is not a problem for the normal MS operation since leases and certificates to change the inactivity status of members can be signed by only  $f_{MS} + 1$  MS replicas. However, it may be important to know the epoch keys of these nodes to encrypt communication during share refreshment protocol (in case some of the remaining replicas fail and do not properly execute the refreshment protocol), and it is also useful to have more epoch keys so we have more possibilities to sign leases. To address this, a node that had not submitted its epoch key may still create one and submit it to the old membership service until this service is stopped.

*6. Sign the new configuration.*



The share refreshment protocol ends after  $2f_{MS} + 1$  new replicas claim they have computed the new shares [23]. At this point, old replicas sign the new configuration, and send it to the new replicas. New replicas request the new configuration from old replicas after a timeout. Once a new replica receives the new configuration, it starts the BFT service for the next membership service.

*7. Discard old epoch shares and epoch keys.*

When the old replicas sign the next configuration, this also serves as an indication that it is safe for the old replicas to discard the old shares, epoch keys, probe database, and the old BFT service state for the MS. Any old MS replica that sees a signed configuration for the next epoch discards all these pieces of state (this can happen, for instance, to a slow replica that did not participate in the epoch transition protocols).

The only state that is not immediately discarded is the old configurations. These are transmitted in the background from the old replicas to the new replicas, in parallel with the remaining protocols. New replicas can request and read each old configuration from a single old MS replica, since these are authenticated and cannot be forged. Once an old MS replica receives an indication from  $f_{MS} + 1$  new replicas that they have transferred all old configurations, it can discard them. Old replicas query new replicas for this assertion after a timeout.

**Slow New MS Replicas**

Another issue is what happens to slow new MS replicas that do not participate in the epoch transition protocols. They may find out about the new epoch later on and need to transfer state from the previous epoch. The solution is that when the new nodes see the new configuration they know the old state has been discarded and they obtain their state from the remaining new MS replicas.

They can produce new shares from information received by  $f_{MS} + 1$  other non-faulty new nodes, a feature that is present in the proactive threshold signature protocols we use [23].

The initial values for the probe database can be obtained directly from the other MS replicas. The slow MS replica will accept the probe database values after it

receives  $f_{MS} + 1$  matching replies. This implies that MS replicas must keep their initial values for the probe database until the epoch ends.

The old configurations can be obtained directly, either from one of the old replicas or from one of the new replicas, depending on who still holds them. The slow new replica can query both old replicas and new replicas for old configurations, and download them from the node to whom it has the fastest connection.

After obtaining this state, the slow new replica can start the service for the new epoch. It may be the case that some operations have already executed on the new service. The BFT protocol ensures that slow MS replicas catch up with all the operations [20].

### 3.4.6 Correctness Conditions of the MS

When we had just one MS node, we required that it never fail. When we had a BFT group, we required that no more than  $f_{MS}$  of them fail during the entire lifetime of the system. Now we have a less constraining requirement:

**Correctness condition for the membership service:** *Each MS replica group must contain no more than  $f_{MS}$  faulty replicas between the moment the epoch prior to the one that corresponds to that particular MS ends, until the moment when the last non-faulty MS replica finishes the new epoch and discards its secret threshold signature shares and its epoch private keys.*

This condition ensures that faulty nodes cannot forge requests, e.g., to remove members, since BFT works when less than one third of the replicas in the group are faulty [20], plus requests that require a vector of signature have at least one signature from a non-faulty replica, and the proactive threshold signature protocol produces valid signatures by combining  $k = f_{MS} + 1$  correctly generated signature shares.

Given this condition, the membership service offers very strong semantics to the replicated services: We ensure that any two nodes that have configurations for the same epoch agree on the system membership. In addition, each node has a certificate that vouches for that configuration that can be shown to any other node in the system, thereby providing a way to bring slow nodes up-to-date.

These strong semantics come at a price: the system may halt (i.e., stop producing new configurations) if we cannot form a set of at least  $2f_{MS} + 1$  old and  $2f_{MS} + 1$  new MS replicas that can communicate when epochs are ending. This would happen, for instance, if a network partition splits the MS in two groups, each larger than  $f_{MS}$ . It has been proven that you need to pay this price for the strong consistency and availability guarantees of the service [37].

## 3.5 Discussion

In this section we discuss the scalability of the design of the MS, and an extended functionality that can be performed by the MS.

### 3.5.1 Scalability

Our approach includes a number of techniques that enable it to scale to large numbers of clients and servers.

Deltas and dissemination trees allow us to communicate information about new configurations efficiently.

The lease mechanism has provisions that make it scalable. First, we expect leases to be relatively long, e.g., an hour or more (and we could make them even longer by making their duration adaptive so they expire right after epochs end, as explained before). Nevertheless, there could be so many clients and servers that the MS is not able to keep up with all the challenges. But it is easy to fix this problem by aggregation: rather than having each client issue its challenges independently, we aggregate client challenges and respond to them all at once. Aggregation is done by servers: clients send challenges to particular servers (selected randomly or based on proximity information); the server collects them for some time period, or until it has enough of them, e.g., a hundred; the server hashes the nonces, and sends the challenge to the MS replicas; the server sends the signed response to the clients, together with the list of nonces that were hashed.

In our scheme each node stores the entire configuration in memory. However, this

is not a concern, even for systems with tens or hundreds of thousands of nodes. If we assume that node identifiers have 160 bits (based on a SHA-1 cryptographic hash function), and we use 1024 bit RSA public keys, then the entire configuration for a system of 100,000 nodes will fit in approximately 14.7 megabytes (which is small compared to current memory sizes).

A related issue is the time it takes for a new client to download the current configuration. (This is not a problem for new servers since they will not be active until the next epoch). Clients can use Merkle trees [69] to download specific parts of the configuration, e.g., a node can learn about nodes in a particular id interval first. Merkle trees can also be used by a reconnecting node to identify the minimum information that needs to be transmitted.

A final scalability issue is having each MS replica probe all system nodes. This is not a problem if the replicated service that uses the MS decides to be slow about declaring unreachable nodes to be inactive (Chapter 8 argues that this is advantageous to reduce the cost of redundancy maintenance).

However, this is a scalability barrier if we need to evict nodes more aggressively. To address this, we designed a simple extension to the MS protocols that allows the probe protocol to scale to even larger systems and/or faster probing. The idea is to use *committees* that serve the purpose of offloading the probing functionality from the MS. These committees can be chosen the same way the MS is chosen, i.e., based on the random number in the configuration. Then each committee can probe a subset of the system members (using the same techniques used by the MS) and report the results to the MS (at the end of the epoch, or earlier if the MS desires).

### 3.5.2 Garbage Collection of Old Configurations

As mentioned, the MS must supply slow servers with old configurations so they can catch up one epoch at a time, and perform state transfer between epochs. Although it might be acceptable for the MS to record this information forever, since all it requires is storage, here we describe how to detect when an old epoch will never be needed again; at that point its storage can be removed.

In the two example replicated services we implemented, a configuration for epoch  $e$  can be discarded when a quorum of  $2f + 1$  replicas in each group in epoch  $e + 1$  have upgraded to epoch  $e + 1$  and finished reading the old state from epoch  $e$ . We believe this can be generalized for applications that any quorum size  $|Q|$ : once a quorum of new replicas have the transferred the state from the previous epoch, the remaining new replicas can obtain that state from the servers in that quorum.

To recognize this condition, the MS needs to understand how replica groups are assigned in the replicated service; otherwise it could not know that it has heard from  $2f + 1$  members of each group. For this purpose we assume the existence of a function *replica-groups* that lists all groups of replicas in the configuration for epoch  $e$ .

In our example services we use successors so it is easy to compute the replica groups for any configuration, and there are only  $N$  of them, where  $N$  is the number of nodes in the system. However, for some assignments it may be too hard to compute the set of replica groups (e.g., if we use a different group for each item, there would be as many different groups as items). Our solution would not work in this case.

The MS maintains, for each epoch corresponding to configurations it holds, the list of replica groups for that epoch, and a count, for each replica group, of how many replicas in the group have entered and completed state transfer to that epoch. This information is piggybacked in the answers to signed ping requests.

The configuration and all associated information for an epoch (and previous epochs) can be discarded when at least  $|Q| = 2f + 1$  replicas for every replica group in that epoch have entered (and completed state transfer to) that epoch or a greater one.

If a node requests a configuration that has been discarded, the MS tells it this. If the node needed the configuration to know which old replica group to transfer the data from, it can skip state transfer (and, if needed, get the data from the  $2f + 1$  correct new replicas that finished state transfer).

Note that, in the case of a replicated MS, we cannot trust a single node that says it is safe to discard a certain epoch. As in other operations, we rely on collecting  $f_{MS} + 1$  signatures and running an operation with those signatures before we discard

an epoch. When the MS replicas change, the new replicas can run an operation on the old service to determine which old configurations they must remember, and then they only request those configurations from old members.

# Chapter 4

## Methodology for Building Dynamic Replicated Services

This chapter describes a methodology for transforming replicated services that work in a static system into services that support membership changes. We also present correctness conditions for replicated services that use the technique.

The methodology presented in this chapter can be applied to different replication protocols. In Chapters 5 and 6 we give two examples where we applied the methodology to build dynamic replicated services.

Some of the steps of the methodology are generic: they must be done in any dynamic, replicated service. For instance, the code running on the client machine must check the validity of the lease, and try to renew it slightly before it expires. Or when a server discovers it skipped an epoch, it must contact the MS to obtain the intermediate configurations so that it can advance one epoch at a time.

We provide support for these common functions in the form of modules that run on clients and servers. These modules handle this service-independent work, and also implement the membership service functionality described in Chapter 3 (e.g., sending probes, replying to probes, generating configurations). These modules are described in Appendix A.

The remainder of this chapter is organized as follows. Section 4.1 presents the methodology for transforming replication algorithms to work in a dynamic setting.

Section 4.2 presents correctness conditions for replicated services that use this methodology.

## 4.1 Methodology for Transforming Replicated Services

The starting point of the methodology is a replicated service designed for a static system, i.e., one that does not support membership changes. We assume the replicated service has the characteristics mentioned in Chapter 2: We assume the service state is partitioned into items, and each item is assigned a replica group that is a subset of the active servers.

We also assume the replicated service includes a proxy on each client. The proxy handles application requests to use the replication service (e.g., to read or write an item) by carrying out a replication protocol. This protocol involves a sequence of one or more *phases*. In each phase the client contacts the replicas responsible for an item and gets back replies from a subset of these replicas. Examples of different protocols are given in Chapters 5 and 6; the protocol in Chapter 5 has two phases, while that in Chapter 6 requires just one phase.

The methodology assumes the existence of configurations that have ordered epoch numbers. These numbers allow nodes to compare the recency of the configurations they hold, as described in Chapter 3.

The goal of the methodology is to transform the replicated service designed for a static system into a service that supports changes to the system membership. Applications using the service should be oblivious to this transformation: They should observe the same service interface, and the new service should provide the same semantics as the static one.

The basic idea is that the protocols must be modified to be aware of epochs. The client-side protocols must ensure that each phase is executed entirely within an epoch. If this cannot occur because some of the servers are in different epochs than the client,



corrective action must take place (the slower parties must upgrade to the “current” epoch) and a phase is retried until it is correctly executed in the “current” epoch.

### 4.1.1 Detailed Steps

In the modified replicated services, clients and servers must maintain the current configuration and the associated epoch number. All messages in the modified protocol are tagged with epoch numbers, so that nodes that communicate agree on what epoch they are in.

The replicated service is designed to run efficiently in the normal case where the epoch does not change while it is carrying out a request. In this case, client and servers agree on the current epoch number, and the replication protocols are run as in the static case.

Additionally, there is work to do to deal with reconfigurations. The modified client protocols (carried out by the client proxy) should work as follows.

1. Send each request for a given item to the set of responsible replicas.
2. If a server replies that it has a lower epoch number than the client, re-send the request to the same server, but include the latest configuration delta (i.e., the configuration certificate) in the request message.
3. If a server replies that it has a higher epoch number than the client, the reply contains the new configuration delta. In this case, the client verifies the new configuration, installs it, and retries the request in the new epoch.
4. Each phase must be executed entirely in the same epoch: if replies from several servers are required to complete the phase, all of them must come from servers in the same epoch. This is important so that the client will not be fooled if an epoch change occurs in the middle of some phase. Sometimes the client may need to re-execute the phase to satisfy this condition, so care must be taken to make the execution of the requests idempotent.

5. Clients need to make sure they hold a valid lease when they process a reply from a server. The lease mechanism ensures that the configuration held by the client is not “too old”, if it were too old the correctness conditions of the replica group that is contacted might no longer hold (see section 4.2).

The server methodology is as follows.

1. If a message received from a client or a server is tagged with an epoch number greater than its own, request the new configuration from that node and upgrade to it as soon as it arrives. Upgrading will trigger state transfer as described in point 4.
2. If a message received from a client or a server is tagged with an epoch number smaller than its own, reject the request and provide the sender with the latest configuration delta.
3. For each epoch, a server knows which items it is responsible for, and it responds to requests only for those items and that epoch. Furthermore, requests are executed only after the server has completed state transfer for that item and that epoch. (This ensures that a response from an honest server reflects everything that happened in earlier epochs and the current epoch.)
4. After the server upgrades to a new epoch, it determines which items it just became responsible for, and performs *state transfer* from the nodes responsible for them in the previous epoch.

In doing so, it can make use of all nodes that were responsible for that item in the previous epoch, even those that are no longer in the current configuration. This is important because the node may have been removed by mistake, and this could lead to a situation where the old replica group would exceed its failure threshold if the new group were unable to make use of the node that was removed.

State transfer is dependent on the replication protocols. However, it must be done in such a way that all operations that happen after the “latest” operation seen by state transfer can no longer be executed in the previous epoch.

In addition, state transfer must let the nodes responsible in the earlier epoch know when their items have been transferred to the next epoch so that they can discard old items. This condition is also dependent on replication protocols, and we further address it in Chapters 5 and 6.

## 4.2 Correctness of the Replicated Service

Our methodology allows the replicated services to tolerate an arbitrary number of server failures provided each replica group meets the required failure threshold (which we assume is in the form “no more than  $f$  failures per replica group”) while the group is “needed”. Now we try to define more precisely what “needed” means. We do this in a way that is not overly restrictive, namely only imposing a small interval during which enough correct servers are required in a replica group.

The system behaves correctly provided each replica group contains no more than  $f$  faulty replicas during the “window of vulnerability” for that replica group. If a replica group is responsible for serving data during epoch  $e$ , the window of vulnerability is the time interval that starts when epoch  $e$  is created, and ends when the replica group is no longer needed. A replica group may be needed for handling requests from clients that are in epoch  $e$ , or for handling state transfer requests for the replicas in epoch  $e + 1$ .

The window of vulnerability is depicted in Figure 4-1. The figure shows that there are two conditions that govern how long nodes are needed after an epoch ends: state transfer and leases. The figure shows the lease expiration time being smaller than the time for the last correct server to finish state transfer, but the reverse may also be true, in which case the window of vulnerability is bounded by the time for the last client lease to expire.

Note that the lease expiration time is measured according to the client’s clock:

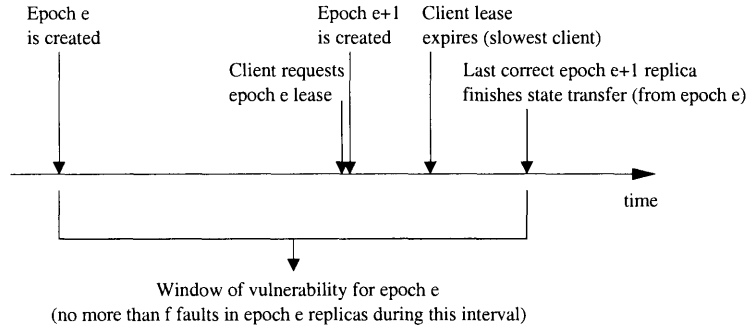


Figure 4-1: The correctness conditions for servers are expressed in terms of a window of vulnerability, which is the time interval depicted in this figure.

we have to take the slowest client into account when determining the window of vulnerability.

We capture the notion of the window of vulnerability in the following correctness condition.

**Replicated Service Correctness Condition.** For any replica group  $g_e$  for epoch  $e$  that is produced during the execution of the system,  $g_e$  contains no more than  $f$  faulty processes between the moment when epoch  $e$  is created, until the later of the following two events: (1) the last non-faulty replica in epoch  $e + 1$  finishes state transfer, or (2) the latest time when the client lease for epoch  $e$  expires at any client  $c$  that accesses data stored by  $g_e$ .

As Figure 4-1 illustrates, the new epoch starts even though leases for the old epoch have not expired. Therefore, holding a valid lease does not mean the node holds the current configuration, only that it holds a recent configuration.

# Chapter 5

## Example I: Dynamic Byzantine Quorum Replication

This chapter presents the first example replicated service we built using the methodology presented in Chapter 4.

This replicated service is based on Byzantine quorum systems [65], but extends this work by supporting a dynamic membership. For this reason we named this system dBQS.

This chapter is organized as follows. Section 5.1 presents an overview of dBQS. Section 5.2 discusses the placement of individual items. Section 5.3 presents the algorithms used by the system. Section 5.4 discusses the correctness of the system.

### 5.1 System Overview

The system we present in this chapter is a replicated data service that provides read and write operations on items (which, in this case, are opaque data objects) that are indexed in a flat namespace.

Object ids are chosen in a way that allows the data to be self-verifying (similarly to previous storage systems like SFS-RO [32] or DHash [25]). dBQS provides access to two types of objects:

- *Content-hash* objects are immutable: once created, a content-hash object cannot

change. The id of a content-hash object is a hash of its contents.

- *Public-key* objects are mutable and contain a version number or *timestamp* that is used to determine data freshness. The id is a hash of the public key used to sign the object. A public-key object includes a header with a signature that is used to verify the integrity of the data, and that covers the timestamp. Public-key objects provide access control (for writing); the scheme could be extended to allow a static set of principals with different public keys to write the same object, by identifying the object by a hash of the public keys of all possible writers, and indicating in the object which key was used in the signature. Extending this scheme to allow changes to the set of writers would probably require an extra level of indirection, and the use of more conventional access control schemes.

Access to public key objects is atomic [54] (or linearizable [42]). This means that all operations on an object appear to execute in some sequential order that is consistent with the real-time order in which operations actually execute. We ensure these semantics despite Byzantine-faulty servers, and membership changes. We assume clients fail by crashing, and we discuss how to extend our algorithms to handle Byzantine clients in Section 5.4.

Content-hash objects have weaker semantics: we only ensure that once the operation that created the object completes, that object is returned by subsequent reads. Having weaker semantics for these objects is a conscious decision, given how we expect them to be used: typically a user publishes their ids (e.g., by modifying a public key object) only after their write completes.

Public-key objects can be deleted (we perform this by overwriting objects with a special *null* value of negligible size), but content-hash objects cannot (since they are immutable). Content-hash objects should be garbage-collected after they are no longer useful. We leave the design of the garbage collection mechanism as future work.

Table 5.1 shows the API that dBQS exposes to applications. We omitted the *delete* operation since it uses the same interface as *put\_s* except we write a special *null*

value. This API is similar to the one supported by storage systems like DHash [25], and applications like file systems have been built using this interface [25, 70].

Function	Description
<code>put_h(value)</code>	Computes the object's id by hashing its contents, and stores it in the system.
<code>put_s(pubkey, value)</code>	Stores or updates a public key object. The value must be signed with the given public key. The object's id is the hash of pubkey.
<code>get(id)</code>	Fetches and returns the object associated with the specified id.

Table 5.1: dBQS application interface.

## 5.2 Object Placement

A key issue is how to partition the objects among the servers. Our implementation uses consistent hashing [48], where node ids have  $m$  bits, and the ids are ordered in an identifier circle modulo  $2^m$ . dBQS stores objects with id  $i$  at the first  $3f + 1$  active members whose identifiers are equal to or follow  $i$  in the identifier space (called the *successors* of id  $i$ ), where  $f$  is maximum number of failures in each replica group; this is how nodes are selected in Chord [90] but any other deterministic selection technique would also be acceptable.

Using consistent hashing has several advantages. It has good load-balancing properties (all nodes are responsible for approximately the same number of objects); it is easy for a node to verify whether it is responsible for a data object; and the work needed to accommodate membership changes is small: only a small number of nodes is involved in redistributing the data.

The current design does not include any access control mechanism, other than the fact that the data is self-verifying. This means that malicious clients (or even malicious servers, as we will see in the next section) can create bogus public key and content-hash objects, and consequently try to exhaust the storage at servers. This attack is subject only to the restriction that it is difficult to target a particular server,

since it requires creating many data objects where the hash of their contents is an id in the interval of the responsibility of that server (or, for public-key objects, create a series of public/private key pairs), which is hard since hash functions are random and not invertible.

## 5.3 Storage Algorithms

This section describes how operations are implemented in our system.

The high-level idea is as follows. We use an existing Byzantine quorum algorithm [66] in the normal case (when there are no reconfigurations). This algorithm provides atomicity for public-key objects in the presence of Byzantine faults with a static replica set [76]. We use a simpler, more efficient version of this algorithm for content-hash objects, since they are immutable. We tag all messages with epoch numbers, and execute each phase of a request in a single epoch. A server accepts a client request only if the epoch number agrees with the current epoch at the server. Otherwise servers notify the client that it is ahead or behind them. When servers hear about a new epoch, they request the corresponding configuration, and then perform state transfer between quorums of different epochs as needed.

The remainder of the section presents the algorithms using pseudocode and some textual descriptions. To eliminate ambiguities, we formally specify the algorithm using I/O automata [64] in Appendix B. We present a correctness proof in Appendix C.

### 5.3.1 Replicated State

Each system node  $n$  (either a client or a server) maintains information about the current configuration, and the current epoch number.

Additionally, *servers* maintain the previous configuration (this is important for purposes of state transfer), a database containing, for each object they store, its id, value, type (content-hash or public key) and, for each public key object, its timestamp and signature (authenticating both the value and the timestamp).

The timestamps form a totally ordered set used to determine the relative order



of operations, and our algorithms require that different clients choose different timestamps. The timestamp space must also contain a minimum element,  $t_0$ . In our implementation timestamps are obtained by concatenating a version number and the client id.

There are other components of the state that we do not explicitly describe here, since they are only used for recording the status of ongoing operations (e.g., to count the number of replies that have been collected thus far). Appendix B describes the state maintained by each node in detail.

### 5.3.2 Client Protocol

Operations must be completed in quorums. These can be arbitrary constructions for dissemination quorum systems [65], but for simplicity of presentation we will assume that at each instant in time there are  $k = 3f + 1$  replicas of every data object and quorums (both read and write) are any subset of those replicas of cardinality  $2f + 1$ , where  $f$  is a threshold set a priori that describes the maximum number of faults in each replica set. The generalization of our algorithms to arbitrary (and reconfigurable) read and write quorums is straightforward.

We now describe how **get** and **put** operations are implemented. We describe these for content-hash objects first, and then for public key objects.

The description does not include retransmissions of client requests, which are important to ensure liveness in the presence of lost or corrupted messages. In our implementation, all requests are retransmitted until the respective reply is received.

Another common feature of the client protocols that we omit is that clients must ensure they hold a valid lease before they accept a reply from a server. When client leases expire, clients must discard new server responses until they renew the lease.

#### Content-hash Put Operation

To create a content-hash object with value  $v_{new}$  and identifier  $x$  (where  $x$  is the hash of  $v_{new}$ ), a client,  $c$ , performs the following sequence of actions.

1. Send  $\langle \text{WRITE-CH}, epoch_c, x, v_{new} \rangle$  to all replicas of the object (the successors of  $x$  in the current configuration held by the client) , where  $epoch_c$  is the epoch number held by the client.
2. Collect replies in a quorum  $Q$ .
3. If all replies are in the form  $\langle \text{WRITE-CH-REPLY}, x \rangle_{\sigma_i}$  (where  $\sigma_i$  represents a correct signature from the sending replica,  $i$ ), the operation is complete. Otherwise, if some reply is of the form  $\langle \text{WRITE-CH-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle$ , send replica  $i$  a  $\langle \text{NEW-CONFIG}, epoch_c, config_c, config\_signature_c \rangle$  message (where  $config_c, config\_signature_c$  are the configuration and configuration signature held by the client) followed by a retransmission of the  $\text{WRITE-CH}$ , remove replica  $i$  from  $Q$ , and try to form a new quorum (wait for more replies). If some reply is of the form  $\langle \text{WRITE-CH-REPLY}, \text{ERR\_UPGRADE\_CONFIG}, next\_epoch, next\_config, \sigma_{cs} \rangle$ , the client verifies the authenticity of  $next\_config$ , upgrades its configuration and restarts the operation. If the configuration or the reply is malformed (e.g., the configuration in the message is old or signatures do not verify), remove the reply from  $Q$  and wait for more replies.

Note that in practice replies from servers that are in a more recent epoch than the client do not contain the entire configuration, but only the list of nodes that were added and deleted, and the client produces the new configuration from the delta and checks the signature. This works in the normal case, when the client is only one epoch behind. We address the case when clients are more than one epoch behind in section 5.3.6.

### Content-hash Get Operation

To read a content-hash object with identifier  $x$ , a client,  $c$ , performs the following sequence of actions.

1. Send  $\langle \text{READ-CH}, epoch_c, x, nonce \rangle$  message to all replicas, where  $nonce$  is a randomly generated number.

2. If a reply in the form  $\langle \text{READ-CH-REPLY}, val \rangle$  is received, where the hash of  $val$  is  $x$ , return  $val$ . Note that in the case of content-hash objects we only need to read from a single replica, and the reply does not need to come from server in the current epoch (e.g., a client can upgrade in the middle of a request and still accept an epoch from an old replica).
3. Otherwise, collect replies in a quorum  $Q$ .
4. If all replies are in the form  $\langle \text{READ-CH-REPLY}, \text{ERR\_NON\_EXIST}, nonce \rangle_{\sigma_i}$  return an error indicating that the object does not exist. Otherwise, handle all remaining cases as in step 3 of the put protocol.

## Public Key Objects

Both get and put operations for public key objects have two phases: a read phase where the highest timestamp is queried (steps 1–3 of the protocols for get and put operations), and a write phase where a new timestamp (or the highest timestamp queried, in case of a read operation) is propagated to a quorum (steps 5–7 of the protocols).

## Public Key Put operation

To put a value  $v_{new}$  to the object with identifier  $x$ , a client,  $c$ , performs the following sequence of actions.

1. Send  $\langle \text{READ}, epoch_c, x, nonce \rangle$  messages to all replicas for  $x$  in the current epoch, where  $nonce$  is a randomly generated number.
2. Collect replies in a quorum  $Q$ .
3. If all replies are in the form  $\langle \text{READ-REPLY}, val_i, ts_i, \sigma_{\langle val, ts \rangle}, nonce \rangle_{\sigma_i}$  (where  $\sigma_{\langle val, ts \rangle}$  is a correct signature from an authorized writer over  $\langle val, ts \rangle$ ), go to step 4.
4. Otherwise, handle all remaining cases as in step 3 of the put protocol for content-hash objects.

4. Choose a timestamp  $ts_{new}$  greater than the highest timestamp it read, and append the client id in the low order bits. Sign the data object and the timestamp, creating  $\sigma_{\langle v_{new}, ts_{new} \rangle}$ .
5. Send  $\langle \text{WRITE}, epoch_c, x, ts_{new}, v_{new}, \sigma_{\langle v_{new}, ts_{new} \rangle}, nonce \rangle$  to all replicas.
6. Collect replies in a quorum  $Q'$ .
7. If all replies are in the form  $\langle \text{WRITE-REPLY}, \text{ACK}, nonce, ts_{new} \rangle_{\sigma_i}$ , the operation is complete. Handle all remaining cases as in step 3 of the content-hash put protocol (retransmitting the WRITE, or restarting the write phase, if necessary).

## Optimizations

As an optimization, the read phase (items 1–3) can be omitted in two situations. First, if there is a single writer (as in some existing applications, e.g., Ivy [70]), the writer can increment the last version number it wrote and use it in the write phase. Second, if clients use a clock synchronization protocol where clock skews are smaller than the time to complete an operation (e.g., if the machines have GPS), they can use their clock readings (concatenated with the client id) as timestamps.

## Public Key Get Operation

For a client,  $c$ , to **get** the value of the object with index  $x$ , it performs the following sequence of actions.

1. Send  $\langle \text{READ}, epoch_c, x, nonce \rangle$  message to all replicas, where  $nonce$  is a randomly generated number.
2. Collect replies in a quorum  $Q$ .
3. If all replies are in the form  $\langle \text{READ-REPLY}, val_i, ts_i, \sigma_{\langle val, ts \rangle}, nonce \rangle_{\sigma_i}$ , choose the  $\langle \text{value}, \text{timestamp} \rangle$  pair with maximum timestamp,  $\langle v, t \rangle$  and go to step 4.
4. Otherwise, handle all remaining cases as in step 3 of the put protocol for content-hash objects.

4. If all replies in  $Q$  agree on the timestamp  $t \neq t_0$ , return  $v$ . If all replies in  $Q$  have  $t = t_0$  then return an error indicating the object does not exist.
5. Otherwise send  $\langle \text{WRITE}, epoch_c, x, t, v, \sigma_{\langle v, t \rangle}, nonce \rangle$  to all replicas.
6. Collect replies in a quorum  $Q'$ .
7. If all replies are in the form  $\langle \text{WRITE-REPLY}, \text{ACK}, nonce, t \rangle_{\sigma_i}$ , the operation is complete and the result is  $v$ . Otherwise, handle all remaining cases as in step 3 of the put protocol for content-hash objects, restarting the current phase if needed.

Note that typically a client will need to carry out only one phase to do a read; the second phase corrects inconsistencies introduced, for example, when some other client fails in the middle of writing.

### 5.3.3 Server Protocol

On the server side, when replica  $i$  receives a request, it validates the epoch identifier. If it is smaller than its current epoch,  $epoch_i$ , it replies  $\langle \text{READ/WRITE-REPLY}, \text{ERR\_UPGRADE\_CONFIG}, epoch_i, config_i, config\_signature_i \rangle$ . If it is larger than its current epoch the reply is  $\langle \text{READ/WRITE-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle$ .

If the epoch number is the same as its own, the server checks if it is responsible for the identifier  $x$  in the current configuration, and ignores the request if not.

Otherwise, normal processing ensues as follows.

In the case of a READ-CH request, if the object with index  $x$  is present in the database and of type content-hash, then return  $\langle \text{READ-CH-REPLY}, val(x)_i \rangle$ , where  $val(x)_i$  is the object value stored in the database. If the object is not present return  $\langle \text{READ-CH-REPLY}, \text{ERR\_NON\_EXIST}, nonce \rangle_{\sigma_i}$ .

In the case of a WRITE-CH request, verify if the identifier of the object matches a hash of its contents. If there is a mismatch, ignore the request. If the request is correct, the server updates its local database to the received value  $v$ . Then the server issues a reply of the form  $\langle \text{WRITE-CH-REPLY}, x \rangle_{\sigma_i}$ .

In the case of a READ request (for a public key object), the server returns  $\langle \text{READ-REPLY}, val(x)_i, ts(x)_i, sig(x)_i, nonce \rangle_{\sigma_i}$ , where  $val(x)_i, ts(x)_i, sig(x)_i$  represent the value, timestamp, and signature for identifier  $x$  stored in the local database. If the object is not in the database, it returns the default value and timestamp:  $\langle v_0, t_0 \rangle$ , where  $t_0$  is the minimum timestamp.

In the case of a WRITE request, the server validates the signature of the object against the data value and its timestamp (this step also verifies that the data was created by an authorized writer). Then, if  $t$  is greater than the previously stored timestamp, the server updates its local database value for  $x$  to the received values  $\langle v, t, \sigma_{\langle v, t \rangle} \rangle$ . Independently of  $t$ , the server issues a  $\langle \text{WRITE-REPLY}, \text{ACK}, nonce, t \rangle_{\sigma_i}$  reply.

### 5.3.4 State Transfer

When a server,  $i$ , in epoch  $e$  receives a valid message with an authenticated configuration for epoch  $e + 1$  it moves to epoch  $e + 1$ , updating its local configuration.

At this point, server  $i$  may discover it is no longer a replica of a subset of the identifier space, in which case it stops serving read and write requests for objects in that subset. Or it may discover that it has become responsible for data objects in a new subset of the identifier space. In this case, state transfer must take place from the old replicas that used to be responsible for these objects. There may be more than one group of replicas for different intervals that need to be transferred. The remainder of this section explains how to transfer state from a single replica group, but this process has to be repeated for all old replica groups.

To transfer old objects from their old group, node  $i$  sends a STATE-TRF-INTERVAL-READ message to the old replicas. The argument for this request is an interval that describes the new subset that node  $i$  has become responsible for. A replica  $j$  that was responsible for (part of) that interval in epoch  $e$  will only execute this request after upgrading to epoch  $e + 1$  (issuing a ERR\_NEED\_CONFIG reply if necessary). Then it issues a STATE-TRF-INTERVAL-READ-REPLY containing the indices of the objects held by the server in that interval. (The request contains a nonce that is signed in

the reply, to avoid replays.)

The new replica gathers all the different indices returned in a quorum of  $2f + 1$  STATE-TRF-INTERVAL-READ-REPLY messages, and requests all of the corresponding objects by issuing a STATE-TRF-OBJ-READ message. Objects are fetched individually, and our implementation maintains a window of size  $W_{st}$  objects that are transferred at a time.

State transfer reads are executed exactly as normal reads with two exceptions. First, a STATE-TRF-OBJ-READ is executed in epoch  $e + 1$  despite the fact that replica  $j$  may no longer be responsible for those indices. Second, there is no write back phase (only steps 1–3 in the read protocol are executed). After receiving  $2f + 1$  replies from distinct previous replicas, replica  $i$  sets its local database values to the value received with the highest timestamp for every identifier  $x$  of correct objects (properly signed) returned in any of the STATE-TRF-READ-REPLY messages, or to any correct content-hash object received.

A problem with this protocol is that malicious replicas can create a large number of bogus objects to waste space at new replicas. As mentioned, this is hard since it would imply creating a large number of self-verifying objects in a particular interval, and this can be avoided with an access control mechanism that is not part of the current design.

Note that between the moment that replica  $i$  upgrades to an epoch, and the moment that replica  $i$  finishes transferring state from the previous epochs, requests for reads and writes in the new epoch must be delayed, and handled only when state transfer for the object being requested concludes. Our implementation minimizes this delay by placing an object that is requested by a client at the beginning of the list of objects to be transferred from old replicas.

### 5.3.5 Garbage-Collection of Old Data

In a large scale system where the responsibility for storing a certain object shifts across different replica groups as the system reconfigures, replicas must delete objects they are no longer responsible for, to avoid old information accumulating forever. In

this section we discuss when it is safe to delete old data.

A new replica sends an ack when it receives a valid response to a state transfer request (the ack and the nonce used in state transfer are signed to avoid replays). An old replica counts these acks and deletes the old objects it is no longer responsible for once it has  $2f + 1$  of them. It explicitly requests acks after a timeout to deal with losses of the original acks.

After it deletes an object, the old replica might receive a state transfer request from a new replica (one it had not heard from previously). In this case its STATE-TRF-INTERVAL-READ-REPLY will not contain any of the deleted objects.

This is safe since non-faulty replicas discard their values only once  $2f + 1$  new replicas indicate they have completed state transfer, and at this point any quorum in the new replica group will contain at least one non-faulty replica that will have transferred the latest object written in the old epoch.

### 5.3.6 Skipped Epochs

State transfer as described above requires that a server know about the previous configuration. But it might not know. For example, the server might have known about epoch 1, missed learning about epoch 2, and then heard about epoch 3.

In this case the server will contact the MS to obtain the missing epoch.

Once the server learns about the configuration for epoch 2, it must do state transfer into epoch 2 followed by state transfer into epoch 3. It may happen that the MS informs the server that epoch 2 is no longer active, in which case the server skips state transfer, and upgrades directly to epoch 3. In general, it is safe to skip state transfer from epoch  $e$  to epoch  $e + 1$  if the MS informs the server that epoch  $e$  is inactive. This is because a quorum of  $e + 1$  replicas is already holding the latest value written in epoch  $e$  or a subsequent write.



## 5.4 Correctness

The system works correctly despite Byzantine-faulty servers, provided each replica group of  $3f + 1$  servers contains less than  $f$  faulty servers in that replica group's window of vulnerability which is defined in Section 4.2.

As far as clients are concerned, the algorithm presented thus far provides atomicity only in the presence of clients that fail benignly (crash failures)<sup>1</sup> and our correctness proof (given in Appendix C) assumes that clients follow the protocol.

A malicious client can cause the system to malfunction in several ways. For instance, it can write different values to different replicas with the same timestamp, thereby leaving the system in an inconsistent state where subsequent reads could return different values; or it can try to exhaust the timestamp space by writing a very high timestamp value.

However, it is not clear what are the desirable semantics in the presence of malicious clients, since these problems do not seem worse than what cannot be prevented, namely, that a dishonest client can write some garbage value, or constantly overwrite the data, provided it has access to that data.

As future work, we intend to define precisely what should be the desirable behavior of this system with malicious clients, and to extend our protocols so that they meet this specification.

---

<sup>1</sup>under the standard assumption that if a client crashes during an operation it is considered an incomplete operation as detailed in [60]



# Chapter 6

## Example II: Dynamic State Machine Replication

This chapter presents the second example replicated service that uses the methodology presented in Chapter 4. This replicated service, named dBFT, is based on the BFT state machine replication algorithm by Castro and Liskov [20]. dBFT can run on a large number servers, and allows the set of servers that run the service to change over time.

The dBFT service state is divided into items. The granularity of an item is specific to the service that is being run and the way that service is implemented on top of dBFT. For instance, if the service is a distributed file system, we can implement items that correspond to file pages, or files, or even entire file system subtrees.

The dBFT service allows arbitrary operations on individual items (unlike the previous example service that only supported blind reads and writes). For instance, we can update an item based on its previous state. Our current design does not allow for multi-item operations. This is left as future work.

The remainder of this chapter is organized as follows. Section 6.1 presents an overview of the BFT state machine replication algorithm and library. Section 6.2 explains how dBFT works in the static case. Section 6.3 extends that to work with configuration changes. Section 6.4 discusses state transfer. Sections 6.5 and 6.6 explains how to garbage collect application and BFT replication state from the system.

Finally, Section 6.7 discusses how to extend BFT to work with a larger number of clients.

## 6.1 BFT State Machine Replication Overview

This section provides a brief overview of the practical Byzantine fault tolerance replication algorithm [17, 20, 21] and BFT, the library that implements it. We discuss only those aspects of the algorithm that are relevant to this thesis; for a complete description, see [17].

### 6.1.1 Algorithm Properties

The algorithm is a form of state machine replication [53, 89]: the service is modeled as a state machine that is replicated across different nodes in a distributed system. The algorithm can be used to implement any replicated service with a state and some operations. The operations are not restricted to simple reads and writes; they can perform arbitrary computations.

The service is implemented by a set of replicas  $R$  and each replica is identified using an integer in  $\{0, \dots, |R| - 1\}$ . Each replica maintains a copy of the service state and implements the service operations. For simplicity, we assume that  $|R| = 3f + 1$  where  $f$  is the maximum number of replicas that may be faulty.

Like all state machine replication techniques, this algorithm requires each replica to keep a local copy of the service state. All replicas must start in the same internal state, and the operations must be deterministic, in the sense that the execution of an operation in a given state and with a given set of arguments must always produce the same result and lead to the same state following that execution.

This algorithm ensures *safety* for an execution provided at most  $f$  replicas become faulty during the entire system lifetime, or within a window of vulnerability of size  $T_v$  (if we employ proactive recovery). We do not intend to use proactive recovery in our services, since we are overlapping many functions in each server.

Safety means that the replicated service satisfies linearizability [44]: it behaves

like a centralized implementation that executes operations atomically one at a time. A safety proof for a simplified version of the algorithm using the I/O automata formalism [60] is sketched in a technical report [19].

The algorithm also guarantees liveness: non-faulty clients eventually receive replies to their requests provided (1) at most  $f$  replicas become faulty within the window of vulnerability  $T_v$ ; and (2) denial-of-service attacks do not last forever, i.e., there is some unknown point in the execution after which all messages are delivered (possibly after being retransmitted) within some constant time  $d$ , or all non-faulty clients have received replies to their requests.

## 6.1.2 Algorithm Overview

The algorithm works roughly as follows. Clients send requests to execute operations to the replicas and all non-faulty replicas execute the same operations in the same order. Since operations are deterministic and start in the same state, all non-faulty replicas send replies with identical results for each operation. The client waits for  $f + 1$  replies from different replicas with the same result. Since at least one of these replicas is not faulty, this is the correct result of the operation.

The hard problem is guaranteeing that all non-faulty replicas agree on a total order for the execution of requests despite failures. A primary-backup mechanism is used to achieve this. In such a mechanism, replicas move through a succession of configurations called views. In a view one replica is the primary and the others are backups. Replicas are numbered sequentially, from 0 to  $3f$ , and the primary of a view is chosen to be replica  $p$  such that  $p = v \bmod |R|$ , where  $v$  is the view number and views are numbered consecutively.

The primary picks the ordering for execution of operations requested by clients. It does this by assigning a sequence number to each request. But the primary may be faulty. Therefore, the backups trigger view changes when it appears that the primary has failed to propose a sequence number that would allow the request to be executed.

To tolerate Byzantine faults, every step taken by a node in this system is based on obtaining a certificate. A certificate is a set of messages certifying some statement

is correct and coming from different replicas. An example of a statement is: “I accept the assignment of sequence number  $n$  to operation  $x$ .”

The size of the set of messages in a certificate is either  $f + 1$  or  $2f + 1$ , depending on the type of statement and step being taken. The correctness of the system depends on a certificate never containing more than  $f$  messages sent by faulty replicas. A certificate of size  $f + 1$  is sufficient to prove that the statement is correct because it contains at least one message from a non-faulty replica. A certificate of size  $2f + 1$  ensures that it will also be possible to convince other replicas of the validity of the statement even when  $f$  replicas are faulty.

Other Byzantine fault-tolerance algorithms [20, 50, 80] rely on the power of digital signatures to authenticate messages and build certificates. The algorithm we describe uses message authentication codes (MACs) [14] to authenticate all messages in the protocol. A MAC is a small bit string that is a function of the message and a key that is shared only between the sender and the receiver. The sender appends this to the protocol messages so that the receiver can check the authenticity of the message by computing the MAC in the same way and comparing it to the one appended in the message.

The use of MACs substantially improves the performance of the algorithm — MACs, unlike digital signatures, use symmetric cryptography instead of public-key cryptography — but also makes it more complicated: the receiver may be unable to convince a third party that a message is authentic, since the third party must not know the key that was used to generate its MAC.

## Processing Requests

When a replica receives a client request, the first thing it needs to do is verify if the request has already been executed. This can happen if the client did not receive the replies to its request (e.g., because the messages were lost) in which case it must retransmit the request to all replicas. If a replica receives a retransmission, it should just retransmit the reply instead of re-executing the request, since requests are not idempotent.

To distinguish retransmissions from new requests, clients assign their own sequence numbers to each operation they invoke on the service, and tag each request with that sequence number. Additionally, replicas maintain the sequence number and the reply of the last operation executed by every client that used the service. This information is maintained as a part of the service state that is hidden from the applications.

If a request arrives and the primary recognizes it is not a retransmission, it uses a three-phase protocol to atomically multicast requests to the replicas. The three phases are pre-prepare, prepare, and commit. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views.

Figure 6-1 shows the operation of the algorithm in the normal case of no primary faults. In this example replica 0 is the primary and replica 3 is faulty. The client begins by sending a request to the primary, which multicasts it to all replicas in a pre-prepare message. The multicast message proposes a sequence number for the request, and if the remaining replicas agree with this sequence number (meaning that they have not assigned the same sequence number to a different request) they multicast a prepare message. When a replica collects a certificate with  $2f + 1$  matching prepare messages from different replicas (including itself), it multicasts a commit message.

When a replica has accepted  $2f + 1$  commit messages from different replicas (including itself) that match the pre-prepare for the request, it executes the request by invoking an application-specific `execute` upcall that is implemented by the service. This call causes the service state to be updated and it produces a reply containing the result of the operation. This reply is sent directly to the client, who waits for  $f + 1$  replies from different replicas with the same result.

If a client does not get  $f + 1$  matching replies after a timeout, it may be the case that the primary was faulty and never forwarded the client request, so the client multicasts the request to all replicas.

There is an important optimization that improves the performance of read-only operations, which do not modify the service state. A client multicasts a read-only

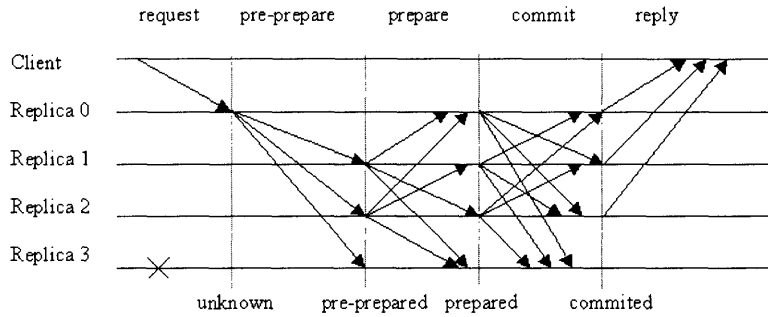


Figure 6-1: BFT algorithm in normal case operation

request to all replicas. The replicas execute the request immediately after some basic checks (e.g., that the client has access). The client waits for  $2f + 1$  replies with the same result. It may be unable to collect this certificate if there are concurrent writes that affect the result. In this case, it retransmits the request as a regular read-write request after its retransmission timer expires. This optimization reduces latency to a single round-trip for most read-only requests.

## Replica State

Each replica stores the service state, a *log* containing information about requests, and an integer denoting the replica's current view. The log records information about the request associated with each sequence number, including its status; the possibilities are: *unknown* (the initial status), *pre-prepared*, *prepared*, and *committed*. Figure 6-1 also shows the evolution of the request status as the protocol progresses.

Replicas can discard entries from the log once the corresponding requests have been executed by at least  $f + 1$  non-faulty replicas, a condition required to ensure that request will be known after a view change. The algorithm reduces the cost by determining this condition only when a request with a sequence number divisible by some constant  $K$  (e.g.,  $K = 128$ ) is executed. The state produced by the execution of such requests is called a *checkpoint*. When a replica produces a checkpoint, it multicasts to other replicas a checkpoint message containing a digest of its state  $d$ , and the sequence number of the last request whose execution is reflected in the state,  $n$ . Then, it waits until it has a certificate with  $2f + 1$  valid checkpoint messages



for the same sequence number  $n$  and with the same state digest  $d$  sent by different replicas. At this point the checkpoint is known to be stable and the replica garbage collects all entries in its log with sequence numbers less than or equal to  $n$ ; it also discards earlier checkpoints.

Creating checkpoints by making full copies of the state would be too expensive. Instead, the library uses copy-on-write such that checkpoints only contain the differences relative to the current state. To enable an efficient copy-on-write mechanism, we partition the service state into a vector of objects of variable size. Each object is identified by its object id in the set  $\{0, \dots, N_{objs} - 1\}$ . To produce checkpoints, the BFT library makes a `get-item` upcall to retrieve a particular object. For slow or corrupt replicas that copied objects from other replicas, the library invokes a `put-items` upcall that updates the state of the application given the values of the objects that are incorrect or out-of-date [84].

## View Changes

The view change protocol provides liveness by allowing the system to make progress when the current primary fails. The protocol must preserve safety: it must ensure that non-faulty replicas agree on the sequence numbers of committed requests across views. In addition, to provide liveness it must ensure that non-faulty replicas stay in the same view long enough for the system to make progress, even in the face of a denial-of-service attack.

View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. A backup is waiting for a request if it received a valid request and has not executed it. A backup starts a timer when it receives a request and the timer is not already running. It stops the timer when it is no longer waiting to execute the request, but restarts it if at that point it is waiting to execute some other request.

If the timer of backup  $i$  expires in view  $v$ , the backup starts a view change to move the system to view  $v + 1$ . It stops accepting messages (other than checkpoint, view-change, and new-view messages) and multicasts a view-change message to all

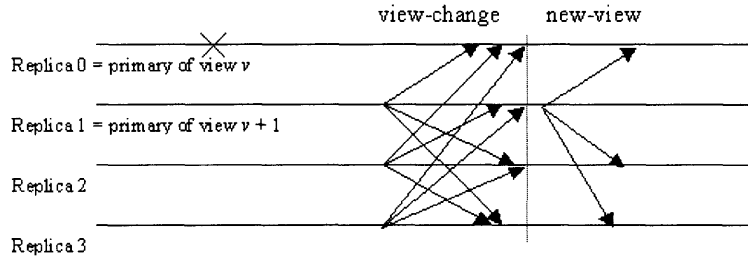


Figure 6-2: BFT view change algorithm

replicas. Figure 6-2 illustrates view changes.

The new primary  $p$  for view  $v+1$  collects a certificate with  $2f+1$  valid view-change messages for view  $v+1$  signed by different replicas. After obtaining the new-view certificate and making necessary updates to its log,  $p$  multicasts a new-view message to all other replicas, and enters view  $v+1$ : at this point it is able to accept messages for view  $v+1$ . A backup accepts a new-view message for  $v+1$  if it is properly signed, if it contains a valid new-view certificate, and if the message sequence number assignments do not conflict with requests that committed in previous views. The backup then enters view  $v+1$ , and becomes ready to accept messages for this new view.

This description hides a lot of the complexity of the view change algorithm, namely the difficulties that arise from using MACs instead of digital signatures. A detailed description of the protocol can be found in [17, 21].

## 6.2 dBFT Algorithms in the Static Case

First, we present how dBFT works in the normal case when clients and servers are in the same epoch, and agree on the system configuration. Section 6.3 presents the extensions to handle reconfigurations.

As in dBQS, each item has a unique id, and we use consistent hashing to determine which servers in the current configuration are replicas for a particular item: these are the successors of the item id in the circular id space.

This already represents a significant departure from the original BFT model [20]. In the original algorithm, there is a single, static replica group. So, for instance, when the primary wants to multicast a protocol message to all backups, it is easy to determine who the backup nodes are. In dBFT, on the other hand, each server is a member of some number of different replica groups in a given epoch (in our case,  $3f + 1$  since we use consistent hashing). Each of these distinct replica groups handles requests for items in a different subset of the id space. And each replica group must implement its own instance of the BFT protocol, with distinct BFT internal state (e.g., message logs) and different service states (e.g., files and directories).

We call each local instance of BFT a *BFT-instance*( $e, i$ ), where  $e$  is the epoch number and  $i$  a “BFT replica number”. The replica number represents the offset in one of the  $3f + 1$  groups that this node is running in (e.g., for the group where it is the lead,  $i = 1$ , and so on). The reason we also include an epoch number when referring to an instance of a BFT replica is that, as will be discussed, when we change epochs, we continue to run BFT instances for earlier epochs as well. So, in general, each dBFT server runs a matrix of BFT instances, with  $e$  identifying the row, and  $i$  identifying the instance within the row.

When a dBFT server receives a message, it needs to forward the message to the appropriate *BFT-instance*. To do this we augment the BFT protocol to include two extra fields that are sent in every message. These two fields are the epoch number and replica number that together identify the remote *BFT-instance* that should handle the message.

Each dBFT server runs a wrapper (we refer to this as the *dBFT-wrapper*) that handles incoming messages, analyzes the epoch number and the BFT replica number, and forwards the message to the appropriate *BFT-instance*.

This architecture is shown in Figure 6-3.

### 6.2.1 Client Protocols

On the client side, there is a client proxy that runs the client side of the dBFT protocols. This is similar to the client proxy for normal BFT, but it needs to be

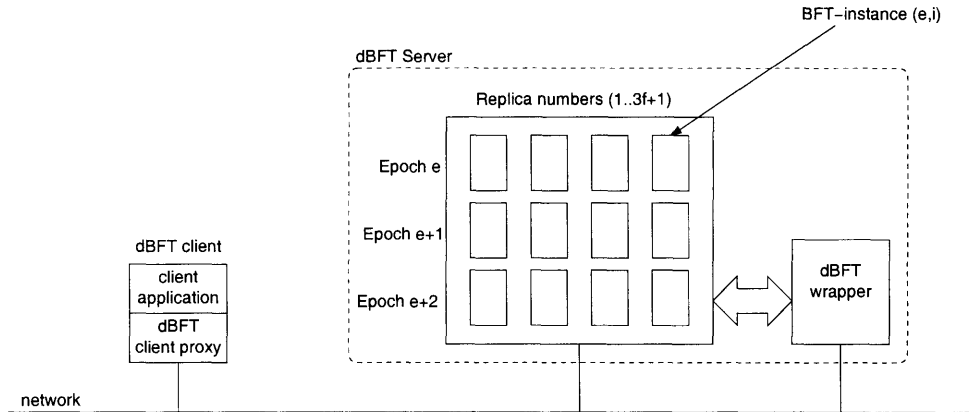


Figure 6-3: dBFT system architecture. Clients run the client application on top of a dBFT client proxy that implements the client-side protocols. Servers run a dBFT wrapper that handles incoming messages, and a matrix of BFT-instances where rows represent different epochs, and columns represent different replica groups that the server is part of in the same epoch. Clients and servers are connected by an unreliable network.

modified in several respects. It must maintain the current configuration, and it must be aware of the existence of multiple groups. The dBFT client proxy knows what item id is associated with the operation it is executing; it determines which replicas are responsible for that id, and sends the requests to these replicas. The new client proxy must also tag each outgoing message with the epoch number and the BFT replica number.

As in dBQS, a client may obtain a reply from a server indicating that client and server are not in the same epoch. The format of these messages and the way they are handled are similar to what happened in that system. If the server is behind, it will reply with a special `ERR_NEED_CONFIG` message, and in this case the client forwards the latest configuration delta to the server and retries the request. In case the client is behind, it will receive a reply of the form  $\langle \text{NEW-CONFIG}, epoch, config\text{-}delta, config\text{-}signature \rangle$ . This reply can come in the form of an individual message from a server or from a normal reply to a BFT request. In this case the client verifies the authenticity of the new configuration, moves to the new epoch and retries the operation in the new epoch.

Finally, the client proxy must check if it holds a valid lease for epoch  $e$  before it

accepts messages for that epoch.

## 6.3 Configuration Changes

Now we move our attention to how servers handle configuration changes. We begin by describing how servers know about configuration changes, then we explain the actions that consequently take place.

### 6.3.1 Delimiting Epoch Changes

The main challenge is to ensure correct serialization between operations in the old epoch and operations in the new epoch. For this purpose, we use the properties of state machine replication itself: we do this by running a special CHANGE-EPOCH state machine operation that delimits the end of each epoch. (Later we describe how this operation is implemented.)

When a *dBFT-wrapper* learns about a new configuration for epoch  $e + 1$ , it invokes the CHANGE-EPOCH operation on all local *BFT-instance*( $e, i$ ),  $i = 1, \dots, 3f + 1$ . (To be more precise, the *dBFT-wrapper* runs an operation on a BFT service, where the local *BFT-instance* is one of the replicas that implement the service.) The *dBFT-wrapper* also creates new row of *BFT-instance*( $e + 1, i$ ),  $i = 1, \dots, 3f + 1$ , which will implement the BFT services that will handle client requests in epoch  $e + 1$ .

There are several ways a *dBFT-wrapper* running on epoch  $e$  can be informed about a new configuration for epoch  $e + 1$ :

- By the membership service, e.g., via the multicast of the new configuration.
- By a client that is ahead of the *dBFT-wrapper* and tried to execute an operation in epoch  $e + 1$ . In this case the *dBFT-wrapper* recognizes the client is ahead and issues a direct ERR\_NEED\_CONFIG reply to the client, and the client forwards the new configuration, as explained. The server must verify the new configuration before it accepts it.

- If a CHANGE-EPOCH operation is run on a service that is implemented by a local *BFT-instance*. This is the case when the CHANGE-EPOCH operation was invoked by another *dBFT-wrapper* on the same replica group as the local server. In this case, the *dBFT-wrapper* does not need to invoke the CHANGE-EPOCH operation on the group where that particular *BFT-instance* belongs to.

### 6.3.2 Handling Epoch Changes

For the *BFT-instance* to handle the special CHANGE-EPOCH operation (and other operations we will describe later), we need to extend it by running a *dBFT-special-req-handler* layer between the BFT replication library and the service code. This layer handles the `execute` upcalls that are invoked by the BFT replication library whenever a request is ready to be executed. It recognizes special requests related to the dBFT protocol (because they have a special header) and handles them with dBFT-specific code.

The *dBFT-special-req-handler* layer maintains a special `status` variable that keeps track of the status of the service. This status is initially *active* when the instance is first created. The CHANGE-EPOCH operation takes the configuration delta as argument and when it is executed the *dBFT-special-req-handler* verifies the new configuration, and, if it is correct, it stores the new configuration and changes the `status` state variable to *inactive*.

For the normal requests, the *dBFT-special-req-handler* checks the value of the `status` variable. If it is *active*, the request is forwarded to the application by invoking its `execute` upcall. Otherwise, the request is not passed to the application code and the client receives a reply of the form

`<NEW-CONFIG, epoch, config-delta, config-signature>`,

containing the configuration for the new epoch.

## 6.4 State Transfer

When a new *BFT-instance* for a new epoch is started, it is initialized with the configuration information for its configuration and the previous one, and it starts off with a clean BFT protocol state (e.g., empty message logs). However, it still needs to obtain its initial service state, which consists of the application state (e.g., the file system's files and directories) and the extra BFT service state that is maintained by the state machine (the sequence numbers for the last client requests). This requires that new *BFT-instances* transfer state from the previous epoch.

State transfer takes the form of several special dBFT operations that are handled by the *dBFT-special-req-handler*. The new instance identifies the different replica groups in epoch  $e$  that held the state that it will become responsible for, and, for each of these groups, it performs the following sequence of actions.

First, the new *BFT-instance* invokes the GET-IDS operation, passing as an argument an id range that is being requested. When this operation is executed on the old service, the *dBFT-special-req-handler* verifies if the service is in the *inactive* state, and if so it returns the ids of all the items in the requested range. If the service is still *active*, the *dBFT-wrapper* rejects the operation, issuing a reply of the form  $\langle \text{GET-IDS-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle$ . In this case, the new *BFT-instance* must invoke the NEW-CONFIG operation on the old service, and then retry the GET-IDS. This ensures that the state that is transferred reflects the latest operation executed in the previous epoch.

After receiving the list of item ids that the old service was storing in the range that is being transferred, the new *BFT-instance* makes a series of STATE-TRANSFER operations, one for each item id, passing that id as an argument. These operations are also handled by the *dBFT-special-req-handler*, which invokes a `get-item` upcall on the service to retrieve an item in the service state. The retrieved items are incorporated in the state of the new *BFT-instance* via a `put-items` upcall. The `get-item` and `put-items` upcalls were also already present in the BASE library for the purposes of checkpointing the service state [84]. We identify the part of the state that holds

the last sequence number of operations executed by clients by a special item id, so this state is transferred using the same mechanism. When a new replica receives sequence numbers for operations by the same client from different groups, it merges this information by keeping only the highest sequence number and the respective reply.

Note that `GET-IDS` and `STATE-TRANSFER` operations are executed very efficiently since we can employ the read-only optimization of the BFT protocol that allows these operations to be executed in a single round-trip.

As an optimization, new instances do not need to transfer items that another local *BFT-instance* (for the previous epoch) is holding. That state can be transferred directly from the respective *BFT-instance* for the previous epoch. This is implemented by calling an application-specific `shutdown` upcall when the old *BFT-instance* becomes inactive. This call saves the service state to disk, although in some cases a large part of that state is already on disk and this call only needs to store whatever state is not already in persistent storage. Then a `restart` upcall is invoked on the new *BFT-instance* that reads the service state from the previously created file.

Note that the `shutdown` and `restart` upcalls were already present in the `BASE` library [84] and they were used to shutdown and restart the node during proactive recovery. We only needed to extend these upcalls to include intervals in the id space that need to be stored or read from disk.

Once state transfer completes, the new *BFT-instance* can start executing the server BFT protocol and start handling operations for the new epoch.

## 6.5 Garbage Collection of Old Application State

As in dBQS, *BFT-instances* must delete application state (e.g., files and directories) they are no longer responsible for, to avoid old information accumulating forever. In this section we discuss when it is safe to delete old application state.

The idea is similar to dBQS. When new *BFT-instances* finish transferring state from the old service, they invoke a `DONE-STATE-TRANSFER` operation on the old



service. The *dBFT* layer in the old service records the number of distinct new *BFT-instances* that have executed that operation for each sub-interval of the id space in the state machine. *BFT-instances* in servers that do not require state transfer are automatically counted as having performed that operation. Once the old service has executed  $2f + 1$  DONE-STATE-TRANSFER operations for a given interval, it informs the application that it can discard the items in that interval. All subsequent state transfer operations for objects in that interval will return a special  $\perp$  reply, which indicates that the interval has been discarded from the old replica's state.

When a new *BFT-instance* receives a  $\perp$  reply to a STATE-TRANSFER operation, this means it can obtain that state from the other  $2f + 1$  new replicas, using the normal BFT protocol checkpointing and state transfer mechanism.

## 6.6 Garbage Collection of Old BFT replicas

Eventually we want to stop handling messages for very old epochs so that we can avoid maintaining a large number of *BFT-instances* running at each server.

We can stop the service after all  $3f + 1$  new *BFT-instances* for all intervals execute the DONE-STATE-TRANSFER operation. However, this may never happen, as some of these *BFT-instances* may run in faulty servers and never execute it.

Instead, when a *BFT-instance* executes  $2f + 1$  DONE-STATE-TRANSFER operations for each portion of the entire id interval it was responsible for, it lets the *dBFT-wrapper* know of this fact. The *dBFT-wrapper* can then shut down that *BFT-instance*. If calls directed to that BFT instance arrive after this point, the *dBFT-wrapper* issues a special  $\pm$  reply, indicating the *BFT-instance* has stopped.

If a new BFT instance trying to fetch old state, or a slow old BFT instance (in the same replica group) trying to execute the BFT protocol, receives  $f + 1$   $\pm$  replies, it can conclude that the old BFT group has been deleted. In the case of the new BFT instance, this means it will do state transfer from the other replicas in the current group. In the case of the slow old BFT instance, it means it can delete itself.

If a client sends a message for a BFT instance that has been deleted the *dBFT-*

*wrapper* issues the NEW-CONFIG reply described before.

## 6.7 Scalability of BFT with the Number of Clients

As mentioned, the BFT protocol is not designed to work with a very large number of clients. The main aspect of the protocol that limits this scalability is the fact that BFT replicas store a client-issued sequence number of the last request executed for every client that executed operations in the group, and the respective reply. This information is stored as part of the BFT service state, and it is used when the client retransmits a request (e.g., if messages were lost and the client never got the reply from enough replicas) to distinguish between a new request and the retransmission. In the case of a retransmission, the replica will just re-issue the reply, instead of executing a new request (since requests may not be idempotent).

We would like to modify this in a way that scales to a larger number of clients. Our approach is to recognize very old requests and garbage-collect them. The service state maintains the last sequence number and reply only for a fixed number of clients.

The set of client requests is maintained in FIFO order: when a request arrives for a client that is not in the set then the last client that has used the service is evicted from the set.

The service state also maintains a variable, *glb*, describing the highest client sequence number that was removed from the set.

If we receive a request from a client that is not in the set maintained by the service, we check to see if the sequence number proposed by the client is higher than the *glb*. If so, the request is executed normally, but if not then the service will issue a special “old” reply. In this case the client will not know what happened, but we believe this will be a very unlikely situation if we make the set of client requests big enough.

# Chapter 7

## Implementation

We implemented the membership service presented in Chapter 3 and the two example replicated services presented in Chapters 5 and 6. This chapter presents an overview of these implementations and some optimizations that we employed.

### 7.1 Membership Service Implementation

We implemented a prototype for the MS in C++. Inter-node communication is done over UDP. For cryptographic operations we use the SFS toolkit [68] implementation of a Rabin-Williams public-key cryptosystem with a 1024-bit modulus, and the 160-bit SHA-1 random hash function.

The main MS loop is written in an event-driven structure. Whenever a node becomes an MS replica, it spawns a replica for an MS service that tracks the operations that occur during that epoch. This service is implemented as a BFT service using the publicly available BFT/BASE code [84].

For proactive threshold signatures we have implemented a variation of the APSS protocol [94] that works in asynchronous systems. The details about the implementation are presented in a previous publication [23].

For the multi-party secure coin tossing scheme [77] we implemented a scheme that is simpler than previous approaches (e.g., [16, 77]), but provides weaker security properties. Appendix D describes this scheme and analyzes its security properties.

## 7.2 Implementation of dBQS

We implemented a prototype for dBQS in C++. Our implementation is based on the code for the DHash peer-to-peer DHT built on top of Chord [90]. Inter-node communication is done over UDP with a C++ RPC package provided by the SFS toolkit [68]. Our implementation uses the 160-bit SHA-1 cryptographic hash function and the 1024-bit Rabin-Williams public key cryptosystem implemented in the SFS toolkit. Objects are stored in a Berkeley DB3 database. dBQS and the membership service module run as separate user-level processes and communicate using a unix domain socket. The two processes run at the same priority level.

Our implementation of the protocols described in Chapter 5 makes a simple optimization with respect to what is described. When executing the read phases of the protocols, we do not send an entire copy of the object in the read replies, as described before.

For content-hash objects we only send a small reply indicating that the replica has the object. The client waits for the first reply and asks that server for the entire object. The size of the object may be larger than the maximum UDP size packet that is supported. Therefore we break the download into multiple RPCs for individual fragments. The first reply contains the size of the object, so the client can issue RPCs for all fragments in parallel. If the client is unable to download the entire object from the first replier after a timeout, we revert to a slower download from all the replicas simultaneously.

For public key objects the first RPC requests a signature covering the timestamp of the object and the nonce issued by the client, plus the size of the object. The client then downloads the object from the first replier. In case the client receives a later reply with a higher timestamp, it changes to downloading from the server that issued that reply. Again, the client must revert to the less efficient method of downloading the entire object from all replicas after a timeout.

Another point to note is that the library uses public key cryptography to authenticate replies to READ and WRITE requests. These signatures are in the critical path of

executing the operations and therefore will introduce a relevant latency to the operations (we discuss performance in Chapter 9). These signatures can easily be replaced with MACs [14] as they are only used in a two-party context. Our implementation does not use these because we assumed a very large system where clients may talk to many different servers and many of the client-server interactions will only last for a single request. In this setting the overhead of establishing a shared secret for the use of MACs is not worth the benefits of not using digital signatures.

When we use public key cryptography, a possible optimization (which we did not implement) is to avoid one of the signatures for the `put` protocol for public-key objects by having servers not sign replies in the `READ` phase, and then sign both the acknowledgment and the timestamp of the read phase in the `WRITE` phase.

### 7.3 Implementation of dBFT

For dBFT we extended the BASE/BFT code [84] to be aware of configurations and support a dynamic membership.

As an example service, we implemented a replicated file system that supports membership changes based on BASE-FS [84]. This is an implementation of a BFT service where the replicas of the service run an unchanged version of the NFS file system server, and a small wrapper that calls the NFS server and hides all non-determinism from the behavior of the server. We extended BASE-FS to support a dynamic membership.

For dBFT and the dynamic version of BASE-FS we used MACs to authenticate communication between clients and replicas. This is because in our implementation of the reconfigurable file system we stored an entire file-system subtree as a single item. This means that the client will contact the same replica group for a long period (except when we reconfigure the system), and therefore it can establish a shared secret with the servers and use MACs for the rest of the operations.



# Chapter 8

## The Cost of Membership Dynamics

This chapter evaluates a cost associated with providing highly available replicated services based on a server set that changes during the system lifetime.

This cost arises from the fact that we require some form of data redundancy to obtain availability guarantees. Maintaining redundancy levels with a changing membership implies that we make new copies of the data as nodes join and leave the system. The rate of data movement is an obstacle to the scalability and dynamics of the system, as nodes have a limited bandwidth to spare for maintaining data redundancy.

In this chapter we determine this cost using a simple analytic model, and we determine possible values for the parameters in this model using values collected from measurement studies of different dynamic systems corresponding to distinct deployment scenarios.

The remainder of the chapter is organized as follows. Section 8.1 presents the analytic model. Section 8.2 elaborates on this model to distinguish between temporary disconnections and membership changes. Section 8.3 uses measurements from deployed system to find values for parameters of the model. Section 8.4 analyzes an important optimization: the use of erasure coding. Section 8.5 presents a discussion of issues related to our model.

## 8.1 Basic Model

### 8.1.1 Assumptions

Our model assumes a large, dynamic collection of nodes that cooperatively store the data. The data set is partitioned and each subset is assigned to different nodes using a deterministic function from the current membership to the set of replicas of each block. This is what happens, for instance, in consistent hashing [48], used by storage systems such as CFS [25].

We make a number of simplifying assumptions. The main simplification comes from the fact that we focus only on an average-case analysis. When considering the worse-case accidents of data distribution, there could be situations where, for instance, more redundancy would be required to maintain data availability in the presence of a spike in system dynamics.

We assume identical per-node space contributions. In reality, nodes may store different amounts of data, and maintaining redundancy may require in certain cases more bandwidth than the average bandwidth.

We assume a constant rate of joining and leaving. Worst-case membership dynamics would be a more appropriate figure for determining the maximum bandwidth required to maintain data redundancy.

We assume a constant steady-state number of nodes. A decreasing population requires more bandwidth while an increasing population would require less bandwidth.

### 8.1.2 Data Maintenance Model

We consider a set of  $N$  identical hosts that cooperatively provide guaranteed storage over the network. Nodes are added to the set at rate  $\alpha$  and leave at rate  $\lambda$ , but the average system size is constant, i.e.,  $\alpha = \lambda$ . On average, a node stays a member for  $T = N/\lambda$  (this is a queuing theory result known as Little's Law [10]).

Our data model is that the system reliably stores a total of  $D$  bytes of unique data stored with a redundancy factor  $k$ , for a total of  $S = kD$  bytes of contributed storage.



If the system redundancy is in the form of replication then the parameter  $k$  represents the number of copies of each data object. If, instead, the system uses erasure codes then  $k$  is the expansion factor (the ratio between the sum of the sizes of the fragments that are stored and the size of the original data block). The desired value of  $k$  depends on both the storage guarantees and redundant encoding scheme and is discussed more in the next section. We assume this is a fixed redundancy factor: the system tries to maintain at all times a factor of redundancy among system members of at least  $k$ .

TotalRecall [13] is a storage system that proposes a variable redundancy factor where many copies are created initially and as flaky nodes leave the system the redundancy levels drop, but new copies are not created provided redundancy is still above some threshold. Section 8.5 discusses this optimization and how it relates to our model.

We now consider the data maintenance bandwidth required to maintain this redundancy in the presence of a dynamic membership.

Each node *joining* the system must download all the data that it must serve later, however that subset of data might be mapped to it. The average size of this transfer is  $S/N$ , since we assume identical per-node storage contributions. Join events happen every  $1/\alpha$  time units. So the aggregate bandwidth to deal with nodes joining the overlay is  $\frac{\alpha S}{N}$ , or  $S/T$ .

When a node *leaves* the overlay, all the data it housed must be copied over to new nodes, otherwise redundancy would be lost. Thus, each leave event also leads to the transfer of  $S/N$  bytes of data. Leaves therefore also require an aggregate bandwidth of  $\frac{\lambda S}{N}$ , or  $S/T$ .

In some cases the cost of leaving the system can be avoided: for instance, if we assumed that data is never discarded by nodes who are no longer responsible from certain objects, and that data is immutable, then the following situation would not require redundancy repairs after a leave. Node  $x$  joins the system at a moment when the redundancy levels for object  $o$  are at the desired levels, and makes a new copy of  $o$ , increasing the redundancy for that object. Then  $x$  leaves before the other nodes that were responsible for the same object leave, and this reinstates the desired redundancy

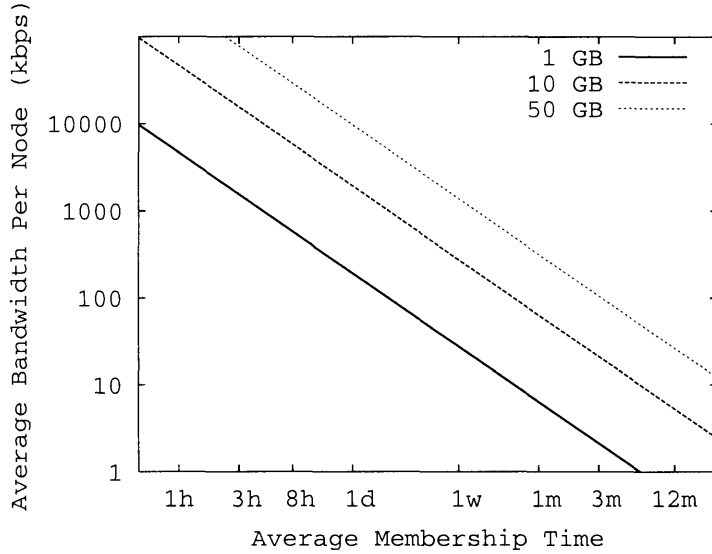


Figure 8-1: Log-log plot of the average data maintenance bandwidth (in kilobits per second), as a function of the average membership lifetime, for three different per-node storage contributions: 1, 10, and 50 gigabytes.

levels.

We will ignore this optimization and therefore the total bandwidth usage for data maintenance is  $B = \frac{2S}{T}$ , or a per node average of:

$$B/N = 2 \frac{S/N}{T}, \quad \text{or} \quad BW/node = 2 \frac{space/node}{lifetime} \quad (8.1)$$

Figure 8-1 shows the consequences of this basic model. We plotted the average bandwidth required to maintain data redundancy as a function of the time members remain in the system for three different per-node storage contributions: 1, 10, and 50 gigabytes. These contributions roughly correspond to 1%, 10%, and 50% of the size of hard drives shipped with commodity desktop PCs sold today. We can see that the maintenance bandwidth can be a serious obstacle to the storage size when systems are dynamic. For instance, for systems where nodes remain in the system for an entire week, and if each node contributes 10 GB to the distributed storage system, the average bandwidth required to maintain data redundancy will be higher than 200 kbps, which is significant.

Another way to state the result is in terms of how stable members must be in order

to bound the average maintenance bandwidth. For instance, to limit this bandwidth to an average of 100 kbps, system members must remain in the system for at least 2, 20, or 100 days, for the three per-node storage contributions above. This makes it hard to build such systems out of volunteer members who may only be willing to “try out” the distributed application.

## 8.2 Distinguishing Downtime vs. Departure

As mentioned in the previous section, long lifetimes are essential to the scalability of cooperative storage. To achieve long memberships, applications should try to make a simple distinction between session times and membership lifetimes (as other authors have noted [12, 82]). This distinction is illustrated in Figure 8-2: A session time corresponds to the duration of an interval when a node is reachable, whereas a membership lifetime is the time from when the node enters the system for the first time until it leaves the system permanently.

Considering only membership lifetimes minimizes the amount of data movement, by avoiding triggering such movement due to temporary disconnections. The side effect of doing this is that nodes will be unavailable for some part of their membership lifetime. We define *node availability*,  $a$ , as the fraction of the time a member of the system is reachable (i.e., it will respond to requests of a client trying to access the data), or in other words, the sum of the node’s session times divided by the node’s membership lifetime.

To maintain data availability in the presence of unavailable system members we must employ appropriate data redundancy schemes. The remainder of this section tries to compute the needed redundancy factor when we use replication. Section 8.4 discusses how the use of erasure codes might improve on the amount of redundancy, and possibly the bandwidth usage of the system.

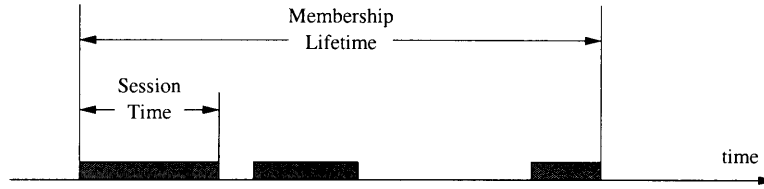


Figure 8-2: Distinction between sessions and lifetimes.

### 8.2.1 Needed Redundancy: Replication

The first redundancy scheme we study is replication, where  $k$  identical copies of each data object are kept at each instant by system members.

The value of  $k$  must be set appropriately depending on the desired per object unavailability target,  $\epsilon$  (i.e.,  $1 - \epsilon$  has some “number of nines”), and on the average node availability,  $a$ . Assuming that node availability is independent and identically distributed (I.I.D.), and assuming we only need one out of the  $k$  replicas of the data to be available in order to retrieve it (this would be the case if the data is immutable and a single available copy is sufficient to retrieve the correct object), we compute the following values for  $\epsilon$ .

$$\begin{aligned}
 \epsilon &= P(\text{object } o \text{ is unavailable}) \\
 &= P(\text{all } k \text{ replicas of } o \text{ are unavailable}) \\
 &= P(\text{one replica is unavailable})^k \\
 &= (1 - a)^k
 \end{aligned}$$

which upon solving for  $k$  yields

$$k = \frac{\log \epsilon}{\log(1 - a)} \quad (8.2)$$

The assumption that one out of  $k$  replicas suffices to retrieve the data may be optimistic. In some cases, namely if the data is mutable and we need to ensure that we retrieve the latest version of the data, we may require more redundancy than the one stated above (e.g., in Byzantine-fault-tolerant replication algorithms we may

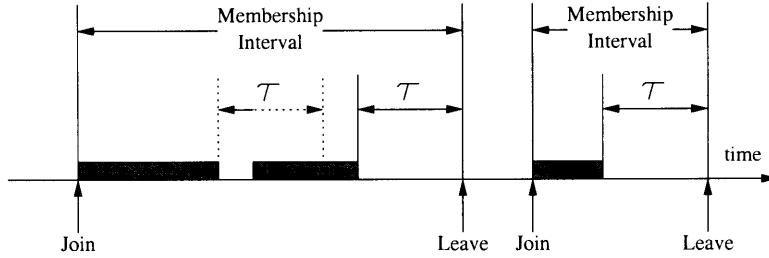


Figure 8-3: Membership timeout,  $\tau$ .

require more than  $\frac{2}{3}$  of the replicas to be available). We do not address this case, but the analysis would be similar to what is presented in Section 8.4.

### 8.3 Detecting Permanent Departures

The problem with this simple model for distinguishing between sessions and membership lifetimes is that it requires future knowledge: applications have no means to distinguish a temporary departure from a permanent leave at the time of a node's disconnection. To address this problem we introduce a new concept of a *membership timeout*,  $\tau$ , that measures how long the system delays its response to failures. In other words, the process of making new hosts responsible for a host's data does not begin until that host has been out of contact for longer than time  $\tau$ , as illustrated in Figure 8-3.

There are several consequences of introducing this membership timeout. A higher  $\tau$  means that member lifetimes are longer since transient failures are not considered leaves. The number of members of the system also increases with  $\tau$ , as more unavailable nodes will be considered members. The average host availability,  $a$ , will decrease if we wait longer before we evict a node from the system.

Translating this into our previous model,  $T$  and  $N$  will now become  $T(\tau)$  and  $N(\tau)$ , and  $a$  will now become  $a(\tau)$ , which implies that  $k$  will become  $k(a(\tau), \epsilon)$  (set accordingly to the equations above). Note that  $a$  decreases with  $\tau$ , whereas  $T$ ,  $N$ , and  $k$  increase with  $\tau$ . By our definition of availability,  $N(\tau)$  can be deduced as  $N(0)/a(\tau)$ . Note that we consider  $a(0) \equiv 1$ , which means that we ignore the period between a

node becoming unavailable and it being evicted from the system membership.

Another consequence is that some joins are not going to trigger data movement, as they will now be re-joins and the node will retain the data it needs to serve after re-joining the system. According to the measurements we will present later this has a minor impact on data movement when we set long membership timeouts (i.e., if  $\tau$  is large enough then there will hardly exist any re-joins) so we will ignore this issue.

Equation 8.1 can therefore be rewritten as

$$B/N(\tau) = 2 \frac{k(a(\tau), \epsilon)D/N(\tau)}{T(\tau)} \quad (8.3)$$

Note that  $B/N(\tau)$  is the average bandwidth used by system members. At any point in time some of these members are not running the application (the unavailable nodes) and these do not contribute to bandwidth usage. Thus we also may want to compute the average bandwidth used by nodes while they are available (i.e., running the application), and to do this we replace the left hand side of Equation 8.3 with  $a(\tau)B/N(0)$  and computing  $B/N(0)$  instead.

### 8.3.1 Setting the Membership Timeout

Setting the membership timeout,  $\tau$ , is not an easy task. A timeout  $\tau$  that is too short may lead to spurious membership changes (and consequent spurious data movement), but if  $\tau$  is too long the host availability may be hampered. The exact dimensions of these problems (and the appropriate values for  $\tau$ ) are highly dependent on the distribution of session times, and thus the expected deployment scenario.

To address this problem, we analyzed three existing traces that represent three possible deployment scenarios for a distributed storage system, and looked at how varying  $\tau$  will affect the behavior of the system. The traces we looked at correspond to the following deployment scenarios.

- Peer-to-peer (volunteer-based) – We used the data collected by Bhagwan et al. on their study of the Overnet file sharing system [12]. Their methodology is as

follows. They crawled the system to find out the identity of 2,400 peers who were members of the overlay at that time. Then they periodically probe each member to see if they are still part of the overlay. The probing is done by looking up the node IDs of the members, to overcome the effects of IP aliasing (nodes behind NATs or using DHCP that may have different IP addresses over time). Each node in the sample is probed every 20 minutes during 7 days.

- Corporate Desktop PCs – We used the data collected by Bolosky et al. [15] on their study of the availability of desktop PCs at Microsoft Corporation. They probed a fixed set of 51,663 machines that participated in the study by sending them ping messages every hour over the course of 35 days.
- Server Infrastructure – This data was collected by Stribling [91] and reflects the reachability of PlanetLab [75] hosts. Unlike the two previous studies that had a centralized prober that verified the reachability and recorded the results, this study measured pairwise ping data between PlanetLab machines. Periodically, ping measurements are taken locally from individual nodes perspective, stored locally, and less frequently archived at a central location. We used results of pings among 186 hosts over the course of 70 days between October and December 2003. Nodes send each other three consecutive pings every 15 minutes. In our analysis of this data, we considered a host to be reachable if at least half of the nodes in the trace could ping it (in at least one of the three attempts).

The methodology used in the Overnet study is somewhat inadequate to the dynamics of the system. The problem is that the membership lifetimes in the Overnet study can be much smaller than the length of the trace. This causes the later parts of the trace to be dominated by stable nodes, since a large fraction of the less reliable nodes will have left the system for good. In terms of the real system these might be replaced with other (possibly also unreliable) nodes, but the trace will not capture that.

The authors of the Overnet study also performed a second experiment, where they crawled the overlay every four hours for a period of 15 days, to discover new members

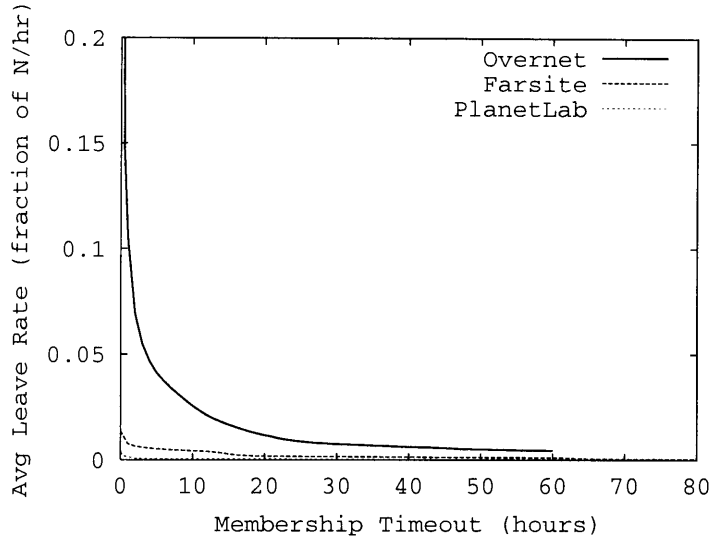


Figure 8-4: Membership dynamics as a function of the membership timeout ( $\tau$ ).

of the system. This experiments gives us a good idea of the real membership lifetime of the system, which we can compare to the lifetimes we deduce from the reachability results. The authors report that each day, new hosts never seen before in the trace comprise over 20% of the system. This implies (using Little’s law [10]) that the average membership lifetime is around 5 days.

Our analysis just looks at the average case behavior of the system. Figure 8-4 shows how increasing the membership timeout  $\tau$  decreases the dynamics of the system. In this case, the dynamics are expressed as the average fraction of system nodes that leave the system during an hour (in the  $y$  axis). Note that by leave we are now referring to having left for over  $\tau$  units of time (i.e., we are referring to membership dynamics, not session dynamics).

As expected, the membership dynamics of the system decrease as we increase the membership timeout, since some of the session terminations will no longer be considered membership changes, namely if the node returns to the system before  $\tau$  units of time.

The most impressive gains in terms of membership dynamics are present in the Overnet trace, where increasing  $\tau$  changes the leave rate from almost half the members leaving each hour, to 0.5% of the members leaving each hour when we set  $\tau = 50$



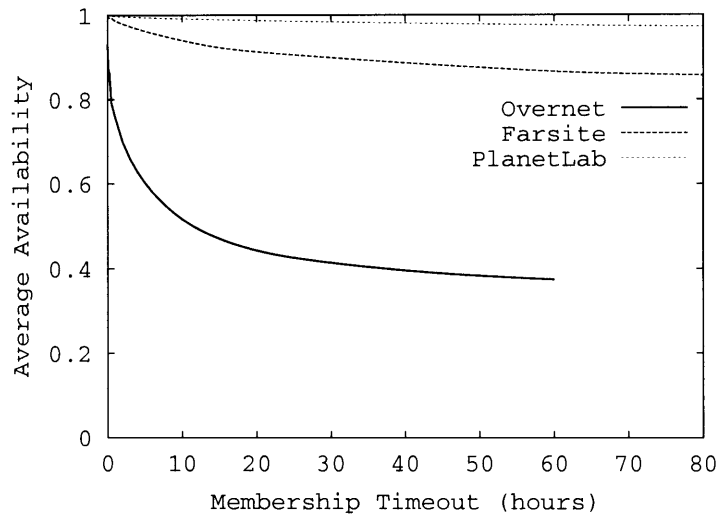


Figure 8-5: Average node availability as a function of the membership timeout ( $\tau$ ).

hours. This reflects the methodology flaw we mentioned above: a leave rate of 0.5% per hour implies a membership lifetime of about 8 days, which is higher than what was deduced from the periodical re-crawling of the system. This is due to the fact that the probing methodology skews the distribution of members to have more stable nodes toward the end of the trace

As mentioned, the second main effect of increasing  $\tau$  is that the node availability in the system will decrease. This effect is shown in Figure 8-5.

Node availability is, as one would expect, extremely high for PlanetLab (above 97% on average), slightly lower for Farsite (but still above 85% on average), and low for the peer-to-peer trace (lower than 50% when  $\tau$  is greater than 11 hours).

An interesting effect from Figure 8-5 is that node availability drops much more quickly in the Overnet trace than in the two other traces. This is the effect of shorter lifetimes in Overnet: waiting before declaring an unavailable node to be out of the membership leads to a more significant contribution to unavailability when there are many short sessions, which is the case in Overnet. On the other hand, a trace like PlanetLab with fewer and longer sessions leads to a slower decrease in availability.

Again, the methodology flaw from the Overnet trace may lead to availability levels that are higher than in reality.

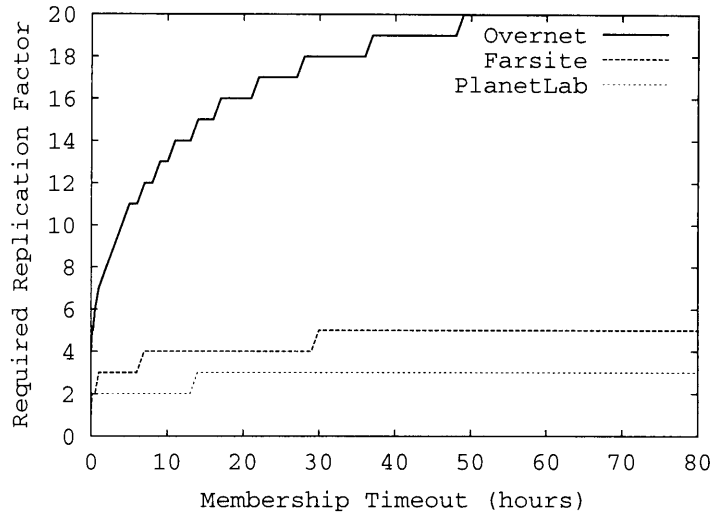


Figure 8-6: Required replication factor for four nines of per-object availability, as a function of the membership timeout ( $\tau$ ).

Note that we did not plot how  $N$  varies with  $\tau$  but this can be easily deduced from the fact that  $N(\tau) = N(0)/a(\tau)$ .

So the next question is how setting  $\tau$  influences the required replication in the system? To answer this we used the availability values of Figure 8-5 in Equation 8.2, and plotted the corresponding replication values, assuming a target average per-object availability of four nines.

The results are shown in Figure 8-6. This shows that Overnet requires the most redundancy, as expected, reaching a replication factor of 20. In the other two deployments replication factors are much lower, on the order of a few units. Note that the figure shows the values obtained from Equation 8.2 rounded off to the next higher integer, since we cannot have fractions of copies.

### 8.3.2 Putting it All Together: Bandwidth Usage

Now we can compute the combined effects of membership timeouts and the required replication factors on the total amount of bandwidth consumed by nodes in a cooperative storage system.

For this we will use the basic equation for the cost of redundancy maintenance

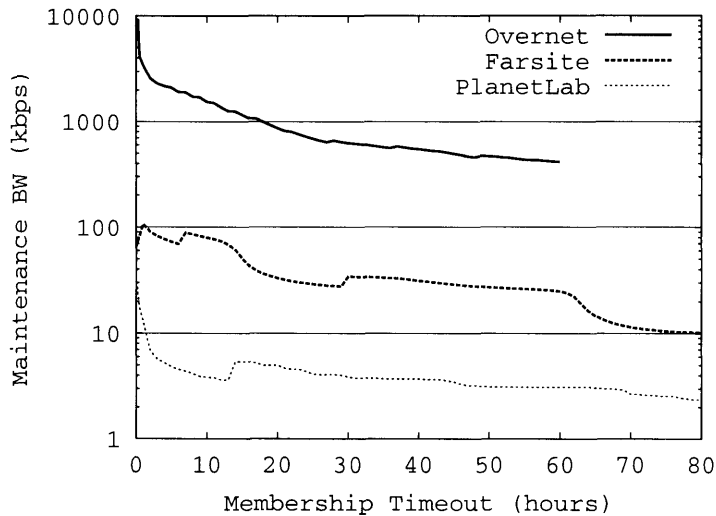


Figure 8-7: Average bandwidth required for redundancy maintenance as a function of the membership timeout ( $\tau$ ). This assumes that 10,000 nodes are cooperatively storing 10TB of *unique* data, and replication is used for data redundancy.

(Equation 8.1) and apply for membership lifetimes the values implied by the leave rates from Figure 8-4 (recall the average membership lifetime is the inverse of the average join or leave rate). We will also assume a fixed number of servers (10,000), a fixed amount of *unique* data stored in the system (10 TB), and the replication factors required for different membership timeouts and consequent availability levels, as computed in Figure 8-6.

Figure 8-7 shows the average bandwidth used for the three different traces and for different values of  $\tau$ . The conclusion is that trying to distinguish between temporary disconnections and permanent departures is important to maintain the bandwidth levels at acceptable values, in any deployment. The downside of using a large membership timeout is that, for a fixed amount of unique data in the system, it requires increased storage to deal with the additional unreachability.

A second conclusion is that even with a large timeout, the bandwidth used in Overnet is problematic (above 200 kbps). Note that Figure 8-7 uses a logarithmic scale in the  $y$  axis, so there is a substantial difference in the bandwidth requirements for the three different traces.

An interesting effect can be observed in the Farsite trace, where the bandwidth

curve has two sudden “steps” (around  $\tau = 14$  and  $\tau = 64$  hours). These corresponds to the people who turn off their machines at night, and during the weekends, respectively. Setting  $\tau$  to be greater than each of these downtime periods will prevent this downtime from generating a membership change and consequent data movement.

## 8.4 Erasure Coding

A technique that has been proposed to save storage and bandwidth is the use of erasure coding [11, 93]. This is more efficient than conventional replication since the increased intra-object redundancy allows the same level of availability to be achieved with much smaller additional storage. With an erasure-coded redundancy scheme, each object is divided into  $b$  fragments and recoded into  $n$  fragments which are stored separately, where  $n > b$ . This means that the effective redundancy factor is  $k_c = \frac{n}{b}$ . The key property of erasure codes is that the original object can be reconstructed from any  $b$  fragments (where the combined size for the  $b$  fragments is approximately equal to the original object size).

We now exhibit the equivalent of Equation (8.2) for the case of erasure coding. Object availability is given by the probability that at least  $b$  out of  $k_c b$  blocks are available:

$$1 - \epsilon = \sum_{i=b}^{k_c b} \binom{k_c b}{i} a^i (1-a)^{k_c b - i}.$$

Using algebraic simplifications and the normal approximation to the binomial distribution (the complete derivation was done by other authors [11]), we get the following formula for the erasure coding redundancy factor:

$$k_c = \left( \frac{\sigma_\epsilon \sqrt{\frac{a(1-a)}{b}} + \sqrt{\frac{\sigma_\epsilon^2 a(1-a)}{b} + 4a}}{2a} \right)^2 \quad (8.4)$$

where  $\sigma_\epsilon$  is the number of standard deviations in a normal distribution for the required level of availability. E.g.,  $\sigma_\epsilon = 3.7$  corresponds to four nines of availability.

Another difference between coding and full replication is how new fragments are

created to deal with the loss of fragments held by nodes leaving the system. In a replication scheme the bandwidth cost associated with this event is just the cost of the new replica downloading a copy of the object from one of the remaining old replicas. With coding, you need to reconstruct the entire object so that you can create the new fragment. To avoid the cost of downloading the entire object (and thus keep the cost of dealing with data loss equal to the amount of data the departing node held) we can employ a simple scheme where one of the nodes responsible for storing fragments of the object will store a complete copy, and this node will be the preferred responsible for creating new fragments as nodes leave the system.

In terms of our model, this would roughly correspond to increasing the coding redundancy factors of Equation 8.4 by one unit (because a complete copy is now maintained alongside with the fragments). With this design, the bandwidth analysis of Section 8.1 is still valid because when nodes leave the system you need to replace the fragments the node held (which is done by the node that holds the copy of the data and only has to push out a new fragment) and the complete copies the node held (which is done by reconstructing a new copy from some of the fragments). Both kinds of reconstruction generate data movement equal to the size of the fragment or data object being replaced, so the analysis still holds.

#### 8.4.1 Needed Redundancy and Bandwidth Usage

To conclude our analysis we will plot the required redundancy factors and the bandwidth usage for maintaining data redundancy using erasure coding, as a function of the membership timeout,  $\tau$ . Figure 8-8 shows the redundancy requirements (i.e., stretch factor) for the availability values of Figure 8-5 for the case when we use coding and set the number of blocks needed to reconstruct an object to be 7 (i.e., we set  $b = 7$  in Equation 8.4). This is the value used by the Chord implementation [26]. Again we assume a target average per-object availability of four nines. The values shown in the figure include the redundancy introduced by the extra copy for restoring fragments (i.e., we added one unit to Equation 8.4).

As shown in Figure 8-8, Overnet still requires more redundancy than the other two

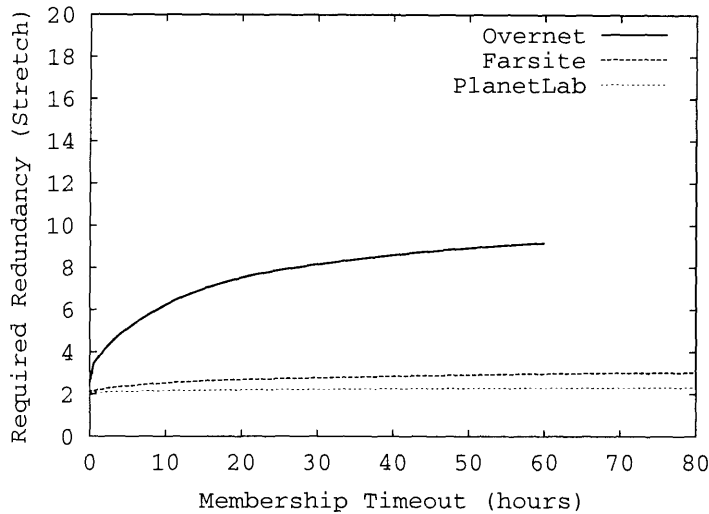


Figure 8-8: Required stretch (coding redundancy factor) for four nines of per-object availability, as a function of the membership timeout ( $\tau$ ).

deployments, but for Overnet coding leads to the most substantial storage savings (for a fixed amount of unique data stored in the system) since it can reduce the redundancy factors by more than half.

Finally, Figure 8-9 shows the equivalent of Figure 8-7 for the case when coding is used instead of replication. The average bandwidth values are now lower due to the smaller redundancy used with coding. However, the bandwidth requirements in an Overnet-like deployment are still too high for a volunteer-based deployment where nodes may be behind slow connections.

## 8.4.2 Replication vs. Erasure Coding: Quantitative and Qualitative Comparisons

The previous section points out that coding can lead to substantial storage and bandwidth savings but it would be desirable to quantify these savings.

Figure 8-10 plots the ratio between the required replication and the required erasure coding stretch factor (i.e., the ratio between equations 8.2 and 8.4) for different server availability levels (assuming server availability is I.I.D.) and for three different per-object availability targets: 3, 4, and 5 nines of availability. In this figure we set

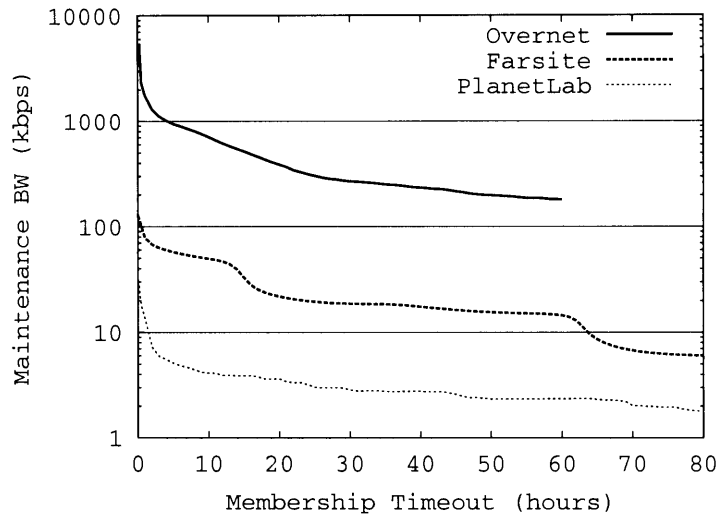


Figure 8-9: Average bandwidth required for redundancy maintenance as a function of the membership timeout ( $\tau$ ). This assumes that 10,000 nodes are cooperatively storing 10TB of *unique* data, and coding is used for data redundancy.

the number of blocks needed to reconstruct an object to be 7 (i.e., we set  $b = 7$  in Equation 8.4), and we included the effects of the extra copy for erasure coding, and of rounding off the replication factors to the next higher integer.

The conclusion is that erasure coding is going to matter more if you store the data in unreliable servers (lower server availability levels) or if you target better guarantees from the system (higher number of nines in object availability). The redundancy gains from using coding range from 1 to 3-fold.

Another advantage of coding (pointed out by a previous paper [26]) is that coding would be useful if we were in the presence of a write-intensive workload where the writer wanted to minimize the time spent in sending out the data.

The redundancy savings from using coding instead of full replication come at a price.

A point against the use of erasure codes is the download latency in a environment like the Internet where the inter-node latency is very heterogeneous. When using replication, the data object can be downloaded from the replica that is closest to the client, whereas with coding the download latency is bounded by the distance to the  $m$ th closest replica. This problem is pointed out with simulation results in a previous

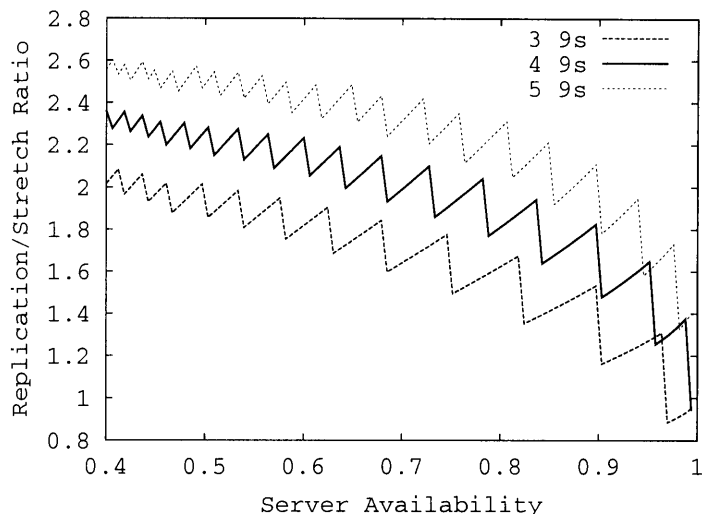


Figure 8-10: Ratio between required replication and required stretch factors as a function of the server availability and for three different per-object availability levels. We used  $b = 7$  in equation 8.4, since this is the value used in the Chord implementation [26].

paper [26].

The task of downloading only a particular subset of the object (a sub-block) is also complicated by coding, where the entire object must be reconstructed. With full replicas sub-blocks can be downloaded trivially. Coding also makes it difficult to perform server-side computations like a keyword search on a block.

A final note about this dichotomy is that erasure coding introduces more complexity into the system (e.g., the task of redundancy maintenance is much more complex, as explained before), and, as a general principle, we believe that complexity in system design should be avoided unless proven strictly necessary. Therefore system designers should question if the added complexity is worth the benefits that may be limited depending on the deployment.



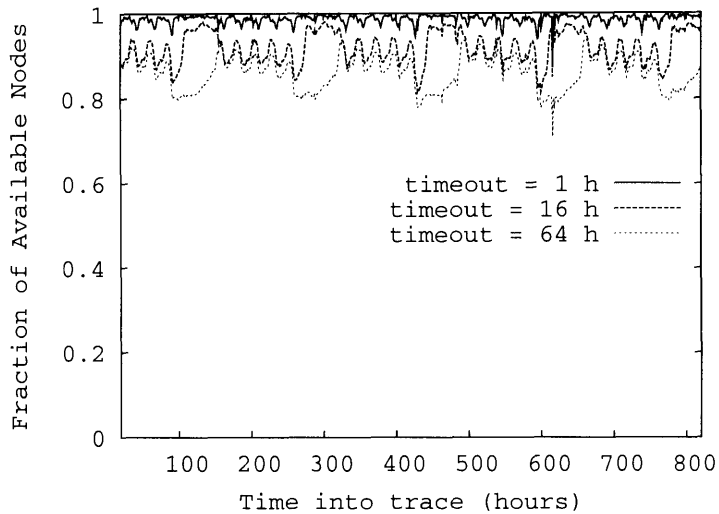


Figure 8-11: Instantaneous availability in the Farsite trace, for three distinct values of the membership timeout ( $\tau$ ).

## 8.5 Discussion

### 8.5.1 Limitations of the Model

A limitation of the previous analysis is that it considered only the average values for node availability, and for the corresponding per-object availability.

The first problem with this is that node availability is not constant, but in fact follows interesting diurnal and weekly patterns. For instance, consider the instantaneous node availability plots for the Farsite trace shown in Figure 8-11. With a 1-hour timeout it is hard to see these patterns, but if we wait for 16 hours before evicting nodes from the system, we can see that all the machines that are turned off at night start to count as system members and the fraction of available nodes goes down at night. With a 64-hour timeout the same happens to machines that are turned off during the weekends. These peaks of unavailability are not captured by the average case analysis of the previous section.

Also, we would like to distinguish more carefully which join and leave events require data movement, since, as mentioned, re-joining or leaving after recent joins created extra redundancy may not trigger data transfer.

Addressing these issues analytically is hard since it would imply having a model for the distribution of node lifetimes, and require a tedious analysis. As future work, we intend to simulate the behavior of a real system using the membership dynamics from these traces.

## 8.5.2 Other Optimizations

In our analytic model for data maintenance we considered two important optimizations that minimize the bandwidth cost: delaying the system's response to failures and using erasure coding.

There are other possible optimizations that could be employed. One of them is *hysteresis* where the system creates a number of replicas (or erasure coded fragments) when the data is first created, and does not try to maintain that number of copies in the system at all times. Instead, it waits until the number of copies falls below a certain minimum threshold, and only then creates new copies (this is proposed in a peer-to-peer storage system called Total Recall [13]).

This technique is useful in settings where the stability of the members is heterogeneous (e.g., peer-to-peer deployments [12, 88]) since it may happen that the initial copies included enough stable members that will ensure the data permanence. The analysis for the amount of initial redundancy required for this is equivalent to, in our analysis, setting the membership timeout to infinity.

There are a few shortcomings with this approach. First, the cost of writing the data (in terms of required time and bandwidth) is increased by the fact that extra copies are necessary to avoid future data movement. Second, you need to be careful about correlated failures: by the time the data level falls below the minimum thresholds, the system may not have enough time to create new copies in case the last remaining nodes leave the system abruptly.

Another characteristic of this design is that it leads to different storage contribution from different nodes: stable nodes will end up with more data as they become the only replicas of old data in the system. This may or may not be considered a positive point.

As future work we intend to study the exact gains of this approach.

### 8.5.3 Hardware Trends

A final discussion point is how hardware trends may influence the future importance of the cost of bandwidth in cooperative storage systems. We could think that the increase in end-user bandwidth might lead to the redundancy maintenance costs becoming less and less relevant. However, the storage trends also have to be taken into account: if the per-node storage contribution grows faster than the per-node bandwidth contribution the cost of redundancy maintenance will become more and more relevant.

To estimate which of these two contributions may increase faster, we did a simple thought experiment: we determined how long it takes to upload your hard drive through your network connection for “typical” home and academic users, and how this figure has evolved over the years.

Year	Disk	Home access		Academic access	
		Speed (Kbps)	Days to send	Speed (Mbps)	Time to send
1990	60 MB	9.6	0.6	10	48 sec
1995	1 GB	33.6	3	43	3 min
2000	80 GB	128	60	155	1 hour

Table 8.1: Evolution of the relative growth of disk space and bandwidth.

Table 8.1 present our results for this thought experiment. The fourth and sixth columns show an ominous trend for the relative growth of bandwidth and disk space: available storage is growing faster than available bandwidth. Therefore, if nodes are to contribute meaningful fractions of their disks their participation must become more and more stable for the bandwidth cost to be reasonable.

Note that “typical” values were intuitively determined by personal experience and not by any scientific means, and are therefore subject to debate.



# Chapter 9

## Evaluation

This chapter presents an experimental evaluation of the three main components of our work. Section 9.1 describes our infrastructure. Section 9.2 presents the evaluation the membership service, while Sections 9.3 and 9.4 focus on the two example replicated services: dBQS and dBFT.

### 9.1 Evaluation Infrastructure

The experiments we describe in this section took place, except otherwise noted, on a wide-area infrastructure consisting of a mix of nodes from different testbeds.

- PlanetLab [75] — Part of the machines in our experiments were nodes from the PlanetLab infrastructure. We used all 296 nodes that were up when we began to run our experiments. These machines were located in approximately 200 sites in four continents. The type of equipment varied considerably (for details about this see <http://planet-lab.org/>). The machines were running version 2.0 of the PlanetLab software that includes a Redhat 9 distribution running a 2.4.22 linux kernel.
- RON [7] — Our wide-area infrastructure also included 22 machines from the RON testbed, scattered over the Internet. Most of these machines had 733 MHz Celeron processors and 256 MB of memory (or similar), and all machines run

FreeBSD 4.7. For more details about the infrastructure see <http://nms.lcs.mit.edu/ron/>

- Local machines — in some cases we wanted to run experiments that would test the system under conditions of extreme load, and, in order to generate this load, we needed to generate a large network traffic. In this case, it is desirable if the load of the machines being tested, and not the wide-area network bandwidth, were the bottleneck of the experiment. In such situations, we populate the system with large number of node running on 10 machines in our local area network. These were Dell Precision 410 workstations with Linux 2.4.7-10. These workstations have a 600 MHz Pentium III processor, 512 MB of memory, and a Quantum Atlas 10K 18WLS disk. All machines were connected by a 100 Mbps switched Ethernet and had 3Com 3C905B Ethernet cards.

## 9.2 The Membership Service

In our evaluation of the membership service we tried to determine how scalable our solution is, and how fast the system can be reconfigured.

### 9.2.1 Scalability

We divide the scalability barriers into two categories: scalability with the number of servers, and with the number of clients.

#### Scalability with the Number of Servers

The main obstacle to scalability with the number of servers stems from the ping protocol, since it is the only component of the system whose complexity grows linearly with  $N$  (the number of nodes in the system).

Remaining costs do not increase as much when  $N$  grows. The cost of running operations for marking nodes as inactive or active, or for explicit addition and removal are proportional only to the number of membership changes (which we expect to

be small or moderate in our deployments) and the number of MS replicas, which is independent of the system size. The cost of the epoch transition protocol also depends on the number of MS replicas and the number of membership changes, with the possible exception of computing signatures of the entire configuration (which is relatively cheap), and of transferring the probe database. The probe database should also be small, since it only contains entries for the nodes that failed the last probe. The percentage of nodes that fail probes is small in stable deployments, as our analysis in Chapter 8 pointed out, so we expect this cost to be low. Additionally there is the cost of propagating new configurations, but our design keeps this cost small by using configuration deltas and multicast trees.

As mentioned in Chapter 3, we can improve on the scalability by delegating the probing responsibility to committees picked among the server nodes. However, to better understand the limits of the design without the committees, we evaluate the scalability of our protocols without this extension.

The first experiment tries to determine how many system nodes a MS replica can ping in a certain interval. To determine this, we monitored a MS replica while varying the rate at which it ping'ed other nodes in a large scale system. (Here we augmented the testbed with about 200 new processes from the local machine set, to avoid saturating wide-area links.) This MS replica was located in our own LAN and was running on an unloaded machine with a 2 GHz Pentium IV processor and 1 GB of memory running Linux 2.4.20. In our implementation of the MS functionality, an active MS replica sends a series of pings, and then sleeps for a predetermined period of time, while awaiting ping replies and other requests. The monitored process verifies a signed nonce for 10% of the ping replies. We measured the scalability of the ping protocol by modifying the number of pings a MS replica sends each time it is awakened. Since we could not control the amount of sleep time at a fine granularity (a limitation of the kernel timer mechanism in Linux/i386 which has a resolution of about 10 ms) we set the sleep time to the minimum we could achieve: 10 ms. This is not an obstacle to measuring the scalability of the system since we can send as many ping packets as desired during a sleep cycle.

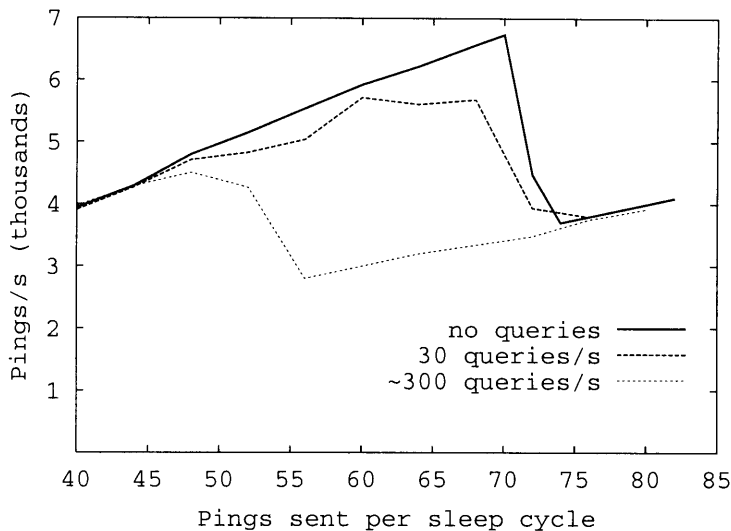


Figure 9-1: Variation of the ping throughput as an MS replica tries to send more probes per sleep cycle. This figure shows this variation for three different levels of activity of a dBQS server co-located with the MS replica.

To determine the scalability of the ping protocol more realistically, the MS replica was associated with an instance of a dBQS server. We repeated the experiment under three different degrees of activity of the dBQS server: when it is not serving any data (which will be the case when the MS runs at a single node or a static replica group), when it is handling 30 queries per second, and when clients saturate the server with constant requests, which leads the maximal number of about 300 queries per second. (We further address the issue of the impact of running MS replicas on the performance of the replicated service in Section 9.3.) Each query requested a download of a 512 byte content-hash block.

Figure 9-1 shows how many pings the MS process could handle per second when we increased the number of pings the MS process sent each time it awakened. The experimental methodology was that, to obtain each point in the graph representing a certain value for the ping throughput, we ran the system until the MS replica handled 400,000 pings, determined the length of time required to do this, and computed the average throughput in number of pings handled per second. The figure shows three lines, corresponding to the different levels of activity in terms of serving data. It shows that the number of pings the node can handle increased linearly with the number of



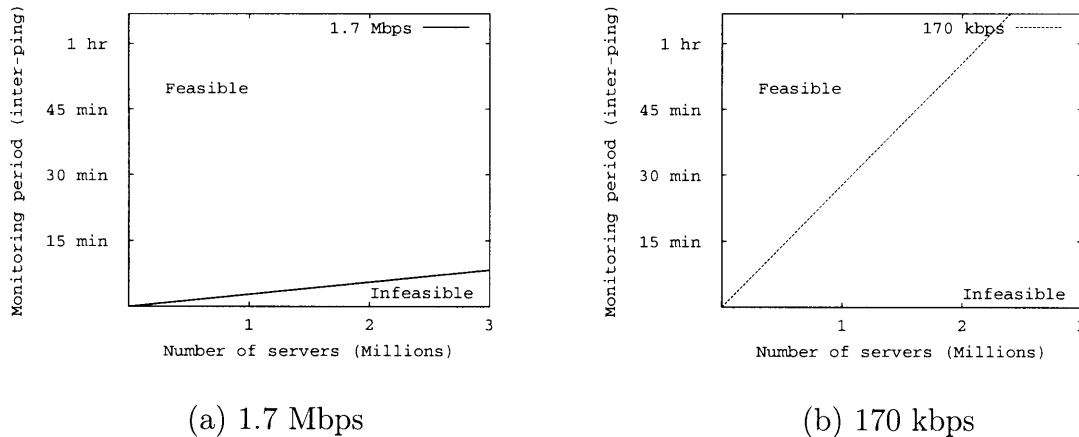


Figure 9-2: Feasibility and unfeasibility regions for different inter-ping arrival rates and system sizes. The lines above divide the deployment planes in two parts: above and to the left of the lines, the system is small enough and the probes are infrequent enough that the probing scheme is feasible with less than the threshold bandwidth. On the other side of the line, the system is so large and the probes are so frequent that the bandwidth usage surpasses the threshold bandwidth. Figure (a) assumes we use 1.7 Mbps for probes, and Figure (b) assumes we use 170 kbps.

pings sent per sleep interval up to almost 7000 pings per second if the node is not serving data, or up to about 5000 pings per second if the node is also serving content. After this point, the node goes into a state of receive livelock, in which the system spends most of its time processing interrupts, and fails to perform other important tasks. This leads to decreasing ping throughput as we try to send more pings.

The ability to process probes is one possible limit to how frequently we send probes, and how many system nodes can the probe protocol handle. However, there is another possible limitation coming from the bandwidth consumed by probing. This bandwidth is relatively high and may be another reason to monitor nodes at a slower rate. Each outgoing packet consists of 8 bytes for the nonce and control information, plus the 28 byte UDP header (we do not consider Ethernet headers since the bandwidth bottlenecks are usually not in the local area). Thus the upstream bandwidth in the MS replicas would be 211 kB/s (or about 1.7 Mbps) for 6000 pings per second. This is a significant bandwidth usage and therefore we may choose to use a slower ping rate for this reason.

If we fix a certain target for the interval between probes to the same machine, the

ping sending frequency (which is directly proportional to the bandwidth usage) determines how many machines a MS replica can probe. Figure 9-2 shows this effect, by depicting the regions where deployments of the monitoring scheme are feasible in the system size / inter-ping arrival rate plane. Threshold lines are shown for two different bandwidth consumption targets: a high bandwidth consumption, corresponding to the maximum ping rate of about 6000 pings per second (or 1.7 Mbps), and a moderate bandwidth consumption of 170 kbps, which corresponds to a ping rate that is 10 times slower. We conclude that even a moderate bandwidth usage of 170 kbps allows for a system with millions of nodes being probed every hour by each MS node. Obviously fewer nodes could be probed if we wanted to use a higher rate.

This analysis ignore the fact that we may want to immediately retry a ping a few times before we declare that a node failed a probe. Again, since most pings are replied immediately, this would not significantly alter the numbers above.

A point to note is that the responsibility of implementing the MS functionality shifts periodically, so this cost is shared in time among the different servers in the system.

It may be desirable to ping very slowly and increase the ping frequency when failures are first detected (but never increase so much that it would cause network congestion). Such a scheme would not have a major impact on the lines of the above figure: we looked into the probability of failed pings in the PlanetLab testbed according to the measurements of Stribling [91] and we concluded that a small fraction of under 1% of the pings to current members fail (assuming we evict failed members in under 11 hours). Therefore even tripling the rate after failed pings would only lead to a 2% increase in the number of pings sent.

The results from Figure 9-2 raise the question of what is an acceptable probe interval? The answer depends on three factors: the desired membership timeout,  $\tau$  (as defined in Chapter 8, this is the time it takes to evict an unreachable node from the set of servers that hold data), the average member availability (i.e., the probability that we cannot contact a system member, for instance due to a temporary disconnection or a network problem), and the false-positive rate for evictions (i.e., the

probability that we remove a member who is still connected).

The idea is that, assuming we probe  $k$  times before we evict nodes from the replicated system, if all  $k$  times the node was only temporarily unavailable, then we have a false positive. Therefore, if we have an average temporary unavailability of  $u$ , the false positive rate is equal to  $u^k$ . This means that the number of probe attempts,  $k$ , should be set to

$$k = \lceil \log_u(\text{false positive rate}) \rceil$$

$$k = \lceil \frac{\log(\text{false positive rate})}{\log u} \rceil$$

Finally, we note that the probe interval should be set to the membership timeout divided by the number of probes required to evict a member, or, in other words, the number of probes is equal to the membership timeout divided by the probe interval.

$$\frac{\tau}{\text{probe interval}} = \lceil \frac{\log(\text{false positive rate})}{\log u} \rceil$$

$$\text{probe interval} = \frac{\tau}{\lceil \frac{\log(\text{false positive rate})}{\log u} \rceil}$$

As we explained in Chapter 8, the membership timeout ( $\tau$ ) and the average node unavailability ( $u$ ) are dependent on the deployment; therefore we used values measured in the analysis of that chapter to determine possible values for the probe interval for different deployments.

Note that we overestimated the node unavailability in Chapter 8 by using both the unavailability due to temporary disconnections and real membership departures. The latter should not be accounted since it does not lead to false positives.

Table 9.1 show possible values for the probe interval for the three different traces and for different values of  $\tau$ . This table assumes we are willing to tolerate a small false positive rate of 0.1%.

Deployment (Trace)	$\tau=6h$	$\tau=12h$	$\tau=18h$
PlanetLab	3 hours	6 hours	9 hours
Farsite	2 hours	4 hours	6 hours
Overnet	45 minutes	1.1 hours	1.5 hours

Table 9.1: Required inter-probe interval in hours for a 0.1% false positive eviction rate for different traces and different values of the membership timeout.

The conclusion is that probing can be done very slowly. For the stable environments, unavailability is uncommon so we can safely evict a node that has only been probed twice for PlanetLab or three times for Farsite. Note that with so few probes we want to repeat a failed probe to account for lost messages: the PlanetLab trace sent 3 pings to a node before it declared a probe attempt to have failed, so it may be desirable to do this as well. For the more unstable environment of Overnet, we need to probe up to 12 times before evicting an unreachable node to ensure it is not a temporary unavailability. However, this still leads to a probe interval of around one hour, which, in the low-bandwidth scenario of Figure 9-2, scales up to around 2 million server nodes.

### Scalability with the Number of Clients

Scalability with the number of clients is dictated by the lease mechanism used to ensure freshness of configurations. If we fix a certain maximum number of leases the MS can grant for each unit of time, this will limit the number of clients and the duration of the respective leases.

To determine the maximum number of leases the MS can grant, we measured the time to produce a digital signature in our local machines. This is a conservative estimate for the time to produce a digital signature since these are five year old machines with a 600 MHz PIII processor. These machines take 21 ms to produce a digital signature, a time we expect to be an upper bound on the time servers will take

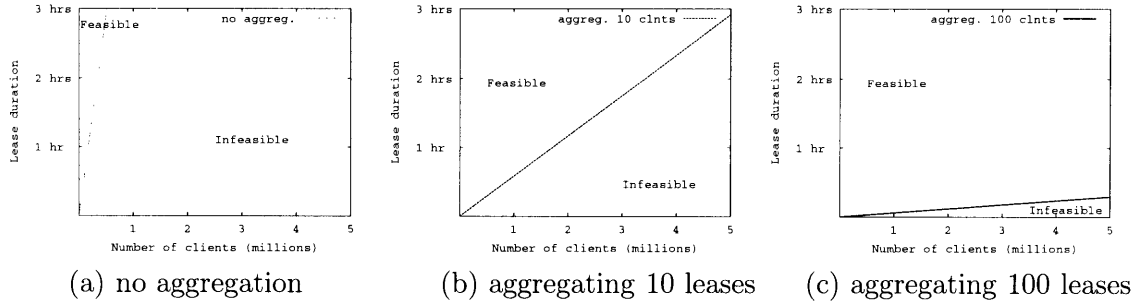


Figure 9-3: Feasibility of client leases in the client population / lease duration space in three cases: (a) without aggregation, (b) aggregating 10 leases at a time, and (c) aggregating 100 leases at a time

to grant leases. Conversely, the inverse of this time is the maximum lease granting frequency: 47.6 leases per second, which implies the feasibility of lease assignment in the client population / lease duration space shown in Figure 9-3. We show feasibility thresholds for three different aggregation values: no aggregation, and aggregating 10 and 100 leases in one signature. These results are conservative for two reasons. First, the signature time was measured using slow machines. Second, this neglects a simple but important point: since we only require  $f_{MS} + 1$  signatures, these requests will naturally be done by different subsets of the replicas (e.g., due to choosing them based on proximity considerations), leading to approximately a 1/3 reduction in replica load in the normal case (no failures).

Nevertheless, the graph shows that aggregation of lease requests is important to allow for large numbers of clients. If we aggregate 100 requests per signature, we can expect to scale to millions of clients with leases shorter than one hour.

## Reconfiguration Frequency

The second part of the evaluation of the MS tried to determine how fast we can reconfigure the system, or in other words, what is the minimum duration for an epoch. We need to choose an epoch duration large enough so that the time it takes to move to the next epoch is a small part of the total duration.

Moving to the next epoch is easy when the MS is implemented at a single node or even at a static group of replicas. But when the MS moves at the end of the epoch,

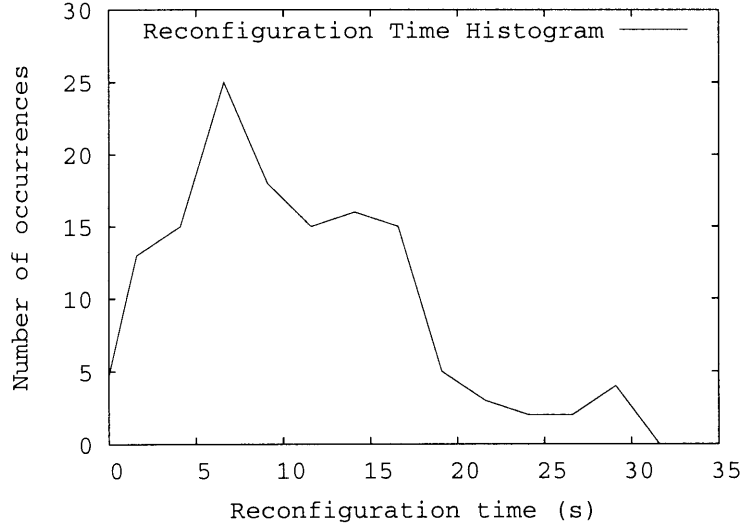


Figure 9-4: Histogram of the local time to reconfigure the system.

we have a number of expensive operations to perform. The most significant of these is proactively resharing the secret [23].

To evaluate the cost of reconfiguration, we conducted an experiment for several days to determine the length of reconfigurations in a wide-area deployment. Again, even though we were limited by the actual size of the testbed (in this case, a few hundred nodes), the main costs are only proportional to the number of changes, not to the number of nodes, and therefore we expect the numbers we obtain to be representative of a real deployment. We ran dBQS in our wide-area infrastructure for several days and measured, for each reconfiguration and each MS replica in the old epoch, the amount of time elapsed between the beginning of the reconfiguration process (invoking the STOP operation on the MS) and its end (ending the share refreshment protocol and signing the next configuration). The MS was running on a group of 4 replicas (i.e.,  $f_{MS} = 1$ ) and was moving randomly among the system nodes.

Figure 9-4 shows a large variation in the time to reconfigure, which is explained by the fact that nodes in the PlanetLab testbed are running many other applications with varying load, and this concurrent activity can affect the performance of the machines significantly. The figure shows that, even in a heterogeneous environment where some machines showed more load than others, most reconfigurations take under

20 seconds to complete. This indicates that cost to reconfigure is not a serious factor in deciding on epoch duration. However, the figure is somewhat misleading because it is done for an MS replica using  $f_{MS} = 1$ . The secret sharing protocol we are using has complexity  $\binom{3f+1}{f}$  [94]; it doesn't scale well as  $f_{MS}$  grows and yet we probably want to run the MS using a reasonably large  $f$ , e.g., 3 or 4. The performance of this protocol as  $f$  grows is evaluated in [23]. This evaluation computes a lower bound on the cost of the share refreshment protocols for fast machines, based on the time those machines take to perform the cryptographic operations involved in the protocol. The conclusion is that with  $f = 2$  this lower bound is over ten seconds, for  $f = 3$  this raises to a few minutes, and for  $f = 4$  a few hours.

## 9.3 Dynamic Byzantine Quorums

This section evaluates the performance of dBQS, the read/write data block service based on reconfigurable Byzantine quorums.

We will first present detailed performance results of different operations without reconfigurations. Then we present an analytic model and an experimental evaluation for the performance of individual operations with reconfigurations. Then we present results that outline the impact of MS operations on this service.

### 9.3.1 Performance with a Static Configuration

The first set of benchmarks represents a controlled experiment that uses a small number of machines on a local area network (which avoids the variability of Internet message delays), and focuses on determining the overhead of public key cryptography and quorum replication for each individual operation in dBFT, and compares the performance of the different types of objects supported by the system.

In this experiment our experimental setup consisted solely of machines from our local set. This allows us to focus on the overhead of cryptography and other local processing steps. The next section presents a set of experiments using our wide-area testbed, which accounts for all the possible overheads.

Object Type	Store Time (ms)	Fetch time (ms)
content-hash	23.9	3.3
public-key (1 phase)	46.2	24.8
public-key (2 phases)	68.4	49.0

Table 9.2: Local area network experiments. Average per object store and fetch time for different systems.

In this experiment we consecutively stored and fetched 256 4kB objects in each of these systems. We repeated the experiment for both kinds of objects. The results are summarized in Table 9.2. The results show the average of three separate attempts. The standard deviations were always below 1.1% of the reported value.

The performance of store operations is roughly proportional to the number of signatures that are in the critical path of the operations (a digital signature takes on average 21 ms to execute on these machines). Content-hash objects only require one signature, by the servers to certify they have stored the object. Public key object operations are slower than content-hash because the client needs to sign the data before storing it, which accounts for the additional latency. We distinguish two kinds of writes. Single-phase writes use the optimization proposed in Chapter 5 of avoiding the read phase in the case of a single writer or synchronized clocks. This leads to only two signatures on the critical path of the operations: the client signing the data, and the servers vouching for storing the data. Two-phase writes require an additional phase for reading the latest timestamp. This phase requires an additional signature in our implementation, which vouches for reading the correct timestamp.

The second column in Table 9.2 shows times for fetching the same number of objects. Again, we see the cost of using public key cryptography to ensure freshness for public key fetches in the presence of Byzantine faults: all repliers must send a signed response containing the current version number and a nonce proposed by the client, since otherwise the client could be tricked into accepting stale data. This is not the case when fetching content-hash objects because these are immutable; thus replies do not need to be signed, and the only additional overhead is for fetching from more replicas.



For two phase reads, we artificially forced the client to write back the value it read (even though it read identical values from all replicas). These are relatively slow operations since they require two signatures in the critical path, but we expect them to be rare, since they occur only if the first phase found an incomplete write.

Furthermore, even though the per-store overhead of dBQS seems significant, it is acceptable when compared to the round-trip latency of Internet communication. For instance, a network round-trip across North America takes more than 70 ms on an uncongested link. This means our system is practical for a wide-area deployment. The next section describes experiments in the wide-area testbed, which shows this effect.

### 9.3.2 Performance During Reconfigurations

We will now analyze the performance of individual operations when clients and servers are not in the same epoch when the operations are initiated. We start with an analytic performance model, which we validate with experimental results.

#### Analytic Model

We define  $d_k$  to be the latency between the client and the  $k$ th most distant replica (we assume there are wide variations among inter-node latencies, as our algorithm is designed for an Internet deployment where replicas are geographically diverse). We define  $l$  to be the local processing time of an action that does not involve producing a digital signature, and  $l_\sigma$  to be the processing time of an action that includes signing. This distinction is important as in practice producing a digital signature takes more than an order of magnitude more time than any other local action. We further distinguish between the time taken by clients to sign data objects,  $l_{\sigma c}$ , and the time taken by servers to sign replies,  $l_{\sigma s}$ , since our experimental evaluation uses different processors at client and server machines.

We simplify the analysis by assuming that at most one reconfiguration occurs during an operation, which is justified by our assumption of a moderate reconfiguration rate. The analysis is for the normal case when there are no node faults. The analysis ignores message pileups and assumes that the probability of a node receiving

Server	Latency (ms)
MIT	21.2
Cornell	21.9
Utah	68.2
UCSD	80.8

Table 9.3: Average round-trip latencies between the client at CMU and different server machines. This reports the average of 20 ICMP pings, the standard deviations were under 0.3 ms.

a message while it is performing a non-signing action is negligible (which is justified by the fact that  $l \ll d$  in practice).

In the case when no nodes reconfigure during the operations, write operations take at most time  $4d_{2f+1} + 2l + 2l_{\sigma_s} + l_{\sigma_c}$ . Read operations take time  $4d_{2f+1} + 3l + 2l_{\sigma_s}$  if the write back phase is required (i.e., when servers do not agree on the current timestamp) or time  $2d_{2f+1} + 2l + l_{\sigma_s}$  in the most common case where servers agree.

Now we consider the case when some of the nodes need to upgrade their configurations. We assume the replicas that are upgrading do not need to transfer state, i.e., they were already replicas in the previous configuration. The case when the replicas need to transfer state leads to a long analysis that we will omit for the sake of brevity.

If a subset  $K$  of the replica set  $R$  is behind the client, the first phase will take the  $2f + 1$ st lowest value of  $\{4d_{c \rightarrow k} + 4l + l_{\sigma_s}, k \in K\} \cup \{2d_{c \rightarrow k'} + 2l + l_{\sigma_s}, k' \in R \setminus K\}$ , where  $d_{c \rightarrow k}$  is the latency between the client and replica  $k$ . The second phase for writes and two phase reads takes an additional  $2d_{2f+1} + l + l_{\sigma_s}$ , plus, in the case of a write operation, the last operation of the first phase becomes a client signature, so we must subtract  $l$  and add  $l_{\sigma_c}$  to the first phase. The total time is the sum of the duration of both phases, as the second phase starts when the first ends.

If  $k$  servers are ahead of the client, there is a race condition in the case where  $k \leq f$ , where if the time to complete the operation with no upgrade at the servers that are not ahead of the client is smaller than  $2d' + 2l$  (where  $d'$  is the closest server ahead of the client), then the operation completes in the time for the normal case. Otherwise, (or if  $k \geq f$ ) the operation will take  $2d^* + 2l$  plus the time for the operation in the normal case, where  $d^*$  is the latency to the closest node that is ahead of the

Reconfiguration Scenario	Write Latency	Predicted Write Latency	Read Latency	Predicted Read Latency
no reconfiguration	227.0	220.4	102.8	100.2
client upgrades	249.5	241.6	126.1	121.4
UCSD and Cornell upgrade	226.4	220.4	103.7	100.2
UCSD and Utah upgrade	298.6	288.6	174.9	168.4

Table 9.4: The performance of read and write operations under different reconfiguration scenarios. The data reflects the average of five trials with a standard deviation of less than 2 ms. All values are in milliseconds (*ms*)

client.

### Experimental Validation

The experimental validation for the performance model used five machines that are part of the our wide-area testbed; the four servers have 733 MHz Celeron processors and 256 MB of memory, while the client machine has a 1.1 GHz PIII processor and 512 MB of memory, and all machines run FreeBSD 4.7. The experiments were run late at night, when network traffic was low and machines were expected to be unloaded. The client machine ran at CMU, and the four servers ran at MIT, UCSD, Cornell, and University of Utah. The average round-trip latency between the client and each server is shown in Table 9.3. Row  $k$  of the table represents  $2d_k$  in our model.

Due to the different processors used in client and server machines, the time to produce a digital signature, which is a good approximation to  $l_\sigma$ , differs among these machines. On the servers, this time is 32 ms, while the client takes on average 20 ms to sign data objects.

Our microbenchmarks consist of a single read or write of a 1 kB data object (this size includes the signature appended to the object). We repeated the operations under different conditions of different nodes changing epochs during the operation. Despite the fact that we changed epochs, we did not change the replica set, to avoid delays related to state transfer. This allowed us to validate the performance model above.

Table 9.4 summarizes the results. The second and fourth column of Table 9.4 show the measured performance of write and read operations, respectively.

In the first reconfiguration scenario, all nodes are in the same epoch initially and there are no epoch changes, so that the write operation completes in two round-trips, and the read operation completes in a single round-trip. Note that, as predicted by our model, we use quorums of  $2f + 1$  replicas so that operations do not need to wait for the most distant replica (in this case, UCSD).

In the second reconfiguration scenario, the client's configuration is initially behind all replicas. In this case, the client upgrades and restarts the operation after contacting the nearest replica (MIT). This adds approximately a round-trip time to the nearest replica to the performance with no reconfiguration (plus an additional smaller latency to verify and install the new configuration).

In the third and fourth reconfiguration scenarios, the replicas at Cornell and UCSD (or Utah and UCSD, respectively) are behind the client and the remaining replicas, initially. This will cause the initial response from these replicas to ask for the new configurations, so that an additional round-trip time must be added to the time to complete the operations at those particular replicas. In the first case, this will not affect the duration of the operation, due to the fact that the latency to Cornell is less than half of the latency to Utah, and therefore Cornell manages to upgrade and reply to the first phase of the operation before a reply from Utah for that phase is received. In the second case, the two most distant replicas need to be upgraded, and this will affect the duration of the operation: The first phase will require an additional round-trip to the Utah replica (the last replica to complete in the quorum) in order for that replica to request the upgrade.

The third and fifth column in Table 9.4 show the values predicted by our performance model when we set  $l_{\sigma_s}=32\text{ms}$ ,  $l_{\sigma_c}=20\text{ms}$ ,  $l=0$ , and  $d$  to the values implied by Table 9.3. This shows that the analysis is correct, yet conservative. Namely, setting  $l = 0$  ignores the cost of some operations that require the verification of a digital signature.

### 9.3.3 Impact of MS operations on Application Performance

Finally, we look at how the superimposing the MS functionality on the dBQS nodes affects the application performance.

We focus on how the performance of dBQS nodes that are assigned the task of being MS replicas degrades due to this task.

The normal case performance within an epoch (when the system is stable) is essentially affected by the ping protocol. (The other cost is that of running state machine operations to add or evict nodes, but these are uncommon. Near epoch boundaries there is the cost of reconfiguring the system but we have seen that epoch transitions are short.) To measure this effect, we ran an experiment that measured how many fetch requests a dBQS node can handle, while we varied the rate at which that node is sending probes.

The setup was similar to the experiments in Section 9.2 where we monitored dBQS server that was serving as an MS replica during that epoch. This server was located in our own LAN and was running on an unloaded machine with a 2 GHz Pentium IV processor and 1 GB of memory running Linux 2.4.20. It was probing nodes on our LAN, and also nodes on the PlanetLab and RON testbeds.

Figure 9-5 shows the results of this experiment. We can see that the fetch throughput decreases from 350 fetches per second to 250 fetches per second as we increase the ping load to near maximal. The degradation is approximately linear, which indicates that we have fine grained control over the tradeoff between probe frequency and the dBQS server performance.

## 9.4 Dynamic Byzantine State Machine Replication

This section presents results of experiments that compare the performance of dBFT with BFT. The experiments were done by running the modified version of the BASE-FS file system [84].

This evaluation differs from the evaluation of dBQS, in that we use MACs to authenticate communication between clients and replicas, whereas in dBQS we used

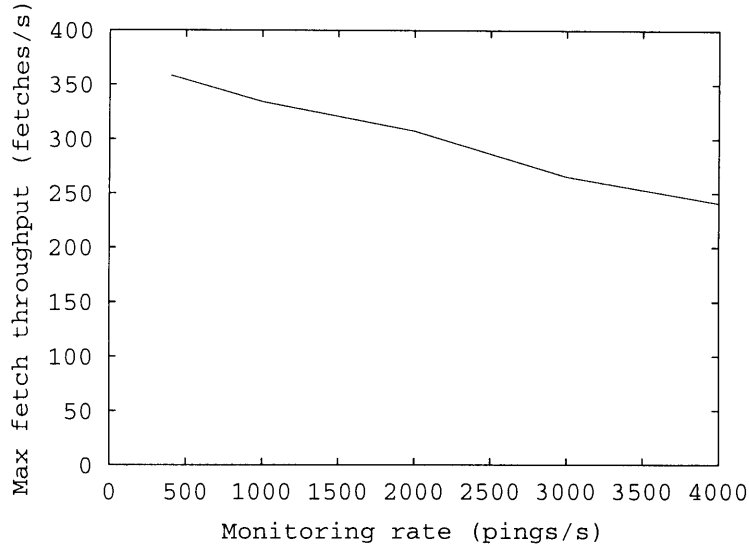


Figure 9-5: Fetch throughput, while varying the ping rate.

public key cryptography. The results from this and the prior section are not meant to be comparable. A comparison between the use of quorums and state machine replication is outside the scope of this thesis.

All experiments ran with one client and four replicas, i.e. we set  $f = 1$ . Scalability of BFT with  $f$  is discussed elsewhere [17]. The results apply to systems with many groups, as only the replica group members have to be contacted by clients to perform operations.

The client ran on the PIII-600 Mhz machine mentioned above, servers ran on Dell PowerEdge 650 servers with dual 3 GHz Intel P4 processors with 2GB of memory. Clients and servers were in different LANs connected by bridges and we emulated multicast by transmitting a series of identical messages to all recipients.

The experiments ran at late hours, when network traffic was low. The machines were always unloaded.

All experiments ran the modified Andrew benchmark [46, 73], which emulates a software development workload. It has five phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data; (4) examines every byte of data in all the files; and (5) compiles and links the files. They ran the scaled up version of the benchmark

	No change	Phase 2	Phase 3	Phase 4	Phase 5
I	2.9	2.8	5.2	2.7	3.0
II	177.7	211.4	184.8	182.1	189.2
III	61.8	62.0	63.5	62.4	62.7
IV	78.2	78.7	74.5	79.2	79.2
V	934.8	946.0	934.1	932.6	946.2
Total	1255.4	1300.9	1262.0	1259.0	1280.4

Table 9.5: Andrew100: elapsed time in seconds

described in [21] where phase 1 and 2 create  $n$  copies of the source tree, and the other phases operate in all these copies. We ran a version of Andrew with  $n$  equal to 100, Andrew100, that creates approximately 200 MB of data.

The benchmark ran at the client machine using the standard NFS client implementation in the Linux kernel with the following `mount` options: UDP transport, 4096-byte read and write buffers, allowing write-back client caching, and allowing attribute caching. All the experiments report the average of three runs of the benchmark and the standard deviation was always below 10% of the reported values.

The results are summarized in Table 9.5. The first column represents the elapsed time for running the Andrew 100 benchmark with a static configuration. The times for the remaining columns correspond to running the same benchmark with only three replicas up in the beginning of the benchmark (the fourth replica was deliberately never started). While the benchmark is still running, during the phase indicated in the header row, the system reconfigures replacing the crashed replica with a new one (running on a fourth machine).

The results show that reconfiguring the replica group during the benchmark does not substantially degrade overall performance: the entire benchmark ran only 0.3–3.6% slower with reconfigurations. Another interesting result is that reconfiguring during a write-intensive phase (phases II or V) causes a bigger slowdown than reconfiguring during a read-only phase (phases III or IV). This is because the new replica, after fetching the initial service state for the second epoch, will be out-of-date with respect to that service, since several updates were executed in the new epoch using the remaining three replicas. Therefore this replica has to fetch state from these

replicas right after it fetched the initial state, causing the other replicas to slow down because they have to provide the missing updates.

Next we tried to breakdown the sources of overhead in the system. There are two main sources of overhead, and one effect that reduces the overhead. The first source of overhead is the time for replicas to upgrade, and start the new service for the new epoch. This cost is due to the fact that replicas have to stop the service, write the old service state to disk, start a new service for the new epoch, and that service needs to read the old state from disk. The second source of overhead is, as explained, the slowdown of the replicas that are supplying state for the node that has become a new replica. The effect that reduces the overhead is the fact that we are moving from a group of three replicas to a group of four replicas, and this allows for BFT to be more efficient by striping complete replies from more replicas: in BFT only one replica has to transmit a complete reply, and the remaining replicas only transmit a hash of the reply, which allows for the overlapping of important transmission costs [20].

To estimate the first source of overhead, we measured the elapsed time between the client receiving a reply indicating it needs to upgrade, and executing an operation in the new configuration, for the different phases when we executed the reconfiguration.

Restart Phase	Time to Upgrade (s)
II	3.4
III	4.1
IV	4.2
V	4.8

Table 9.6: Time to reconfigure. These are the measured times between the client receiving a reply with the new configuration and the client executing an operation in the new configuration, for different phases of the Andrew 100 benchmark when the reconfiguration occurs. Each entry is the average of three reconfiguration times. The standard deviation was under 7% of the average.

Table 9.6 shows this reconfiguration time. This shows that the time for the client to start executing requests in the new epoch was under 5 seconds. This time is somewhat significant, and could be optimized if our implementation would not start a new service, but instead would just modify the internal state of the existing service. How-



ever, assuming that reconfigurations are rare, we believe this overhead is reasonable, and the total running times for the benchmark support this. We also note that the overhead for reconfiguration is small when compared to the other source of overhead (supplying state to the slow replica), especially when the reconfigurations occur in write-intensive phases (II and V).



# Chapter 10

## Related Work

We divide the related work presented in this chapter according to the three contributions of the thesis: the membership service for large scale distributed systems; replication algorithms that support a dynamic membership; and the analysis of the cost of redundancy maintenance on dynamic systems.

### 10.1 Membership Service

Several prior systems have either either assumed the existence of a membership service that detects and proposes membership changes, or have included some form of a membership service. In general, previous approaches are not designed to be very scalable, or do not provide a consistent view of the system membership.

#### 10.1.1 Applications with Membership Changes

Automatic reconfiguration was a goal in many existing systems. Important examples include Petal [57], xFS [8], and Porcupine [87]. These systems include application-specific approaches to deal with membership changes.

Petal [57] is a storage service that provides the abstraction of virtual disks. Petal partitions data among a pool of servers connected by a fast network, and replicates the data for high-availability. Petal can automatically incorporate new servers. This

is achieved by a global state module and a liveness module running on every server. The global state module maps different parts of the state of the system to the current servers that are responsible for them. The global state module maintains its state using the Paxos state machine replication protocol [55], which tolerates up to half of the nodes failing by crashing. Therefore adding a new server to the system is mostly a matter of running operations on that state machine. The liveness module is responsible for detecting and informing other modules of failures, and its operation is not detailed.

xFS [8] is a decentralized file system where multiple machines cooperate as peers to provide the file system service. xFS replicates a “manager map”, containing information about the system membership and the responsibility assignments in the system, at all clients and servers. When the system detects a configuration change, it initiates a global consensus protocol. The consensus algorithm begins by running a leader election protocol (which is not described, but typically involves contacting all system members). The new leader then computes a new manager map and distributes it among the system nodes. The authors do not describe how to tolerate leader failures or concurrent reconfigurations.

Porcupine [87] is a scalable mail server based on a large cluster of commodity PCs. The system provides a membership service that maintains the current membership set, detects node failures, and distributes the new system membership. This service is implemented by all system nodes. Nodes probe their immediate neighbors in the id space to detect unreachability. When any node detects that its neighbor failed a series of probe attempts, it becomes a coordinator for a membership change. The coordinator broadcasts a “new epoch” message with a unique epoch id, and collects replies from all available system nodes. After a timeout period, the coordinator defines the new membership to be those nodes from which it received a reply. In the final round, the coordinator broadcasts the new membership and epoch id to all nodes. As part of this protocol, the coordinator also reassigns the partitioning of the state of the system among the current system members, and broadcasts the new assignments. If two or more nodes attempt to become a coordinator at the same time, the one

proposing the largest epoch id wins. Node additions go through a similar process.

In all these applications, membership changes imply that all nodes agree on that change by running a global consensus protocol. The problem with these approaches is that global consensus does not work in large scale systems.

We took a more principled approach to handling reconfigurations by building a service that tracks membership changes, and separating it from the application design. Furthermore, our configuration management techniques differ from the approaches in these systems because we provide strong security using BFT techniques and because our system is designed to scale better, by avoiding the need for consensus among all system nodes upon reconfiguration, and by batching configuration changes at the end of epochs, among other design choices.

### 10.1.2 Group Communication Systems

The membership service shares the same goals as the group membership modules that were present in many proposals for group communication systems (see [24] for a survey).

Group communication is a means for providing multi-point to multi-point communication, by organizing processes in sets called groups. Processes can send a message to a particular group. The group communication service delivers the message to all the group members. The set of members of the group is allowed to change over time.

These systems are divided into a membership module that maintains a list of the currently active and connected processes in a group, and a reliable multicast service built using the output of the membership module.

Membership modules can be either “primary component” or “partitionable”. In a primary component membership module, views installed by all the processes in the system are totally ordered. In a partitionable one, views are only partially ordered (i.e., multiple disjoint views may exist concurrently).

The specification for the membership module of primary component group communication systems is similar to the specification of our membership service.

Most solutions for the group membership module are not very scalable since they

require all nodes to carry out a consensus protocol for changing the system membership.

An exception comes from the work of Guerraoui and Schiper that proposes a small, separate consensus service that could solve the problem of group membership with fail-stop failures [41]. We improve on this work in several respects: Our work tolerates Byzantine faults, we show how to superimpose the consensus service among the system members, and finally their work does not seem to be implemented.

The only proposals for group communication systems that tolerate Byzantine faults come from the Rampart [79] and SecureRing [50] systems. Adding and removing processes in these systems is a heavyweight operation: all nodes in the system execute a three-phase Byzantine agreement protocol [80], which scales poorly. In contrast, the algorithms used by the membership service were designed to work with thousands of nodes by executing agreement only among a subset of the nodes.

Another problem with these systems is that they must exclude faulty replicas from the group to make progress (e.g., to remove a faulty primary and elect a new one), and to perform garbage collection. For example, a replica is required to know that a message was received by all the replicas in the group before it can discard the message. So it may be necessary to exclude faulty nodes to discard messages. These systems assume that a lack of response from a node implies that the node is faulty, which may not always be true.

To reduce the probability of misclassification, failure detectors can be calibrated to delay classifying a replica as faulty. However, for the probability to be negligible the delay must be very large, which is undesirable in these systems. For example, if the primary has actually failed, the group will be unable to process client requests until the delay has expired, which reduces availability.

Furthermore, misclassification can open an avenue of attack in their system they reduce the group size after a node identified as failed, which is equivalent to saying that they reduce the failure threshold. Therefore an attack that slows down correct replicas can cause the system to be more vulnerable to the existing faulty replicas.

In our system misclassification can also occur, but we reduce its probability by

having a delayed response to failures. Our algorithms do not stall when we delay the response to failures, since we can always make progress despite  $f$  faulty replicas, so we can minimize the occurrence of misclassification without the problems mentioned above. Furthermore, when we remove nodes from the system we are not lowering the failure thresholds, since we maintain the predetermined group sizes, and just redistribute the responsibilities to other system nodes.

### 10.1.3 Peer-to-Peer Systems

Peer-to-peer systems provide several tools for building large-scale applications that support a dynamic membership. We were inspired by these systems in several ways: they also assign responsibility within the system based on node ids and the set of currently available servers, and they share our goals of scalability and automatic reconfiguration.

These systems are based on a lookup layer that locates a set of available nodes responsible for any value in the id space. The operations implemented by applications can be preceded by a lookup step that locates these nodes.

This lookup functionality is implemented using routing protocols: Each node has a limited knowledge of the current set of available members of the overlay. By contacting several nodes in sequence, the requesting node will be able to obtain enough knowledge about the membership to conclude the lookup.

Several systems have been built on top of the lookup layer. The systems that are more closely related to our work are distributed hash tables (DHTs) [25, 40, 59, 78, 86]. These provide a storage substrate with a get/put interface similar to dBQS.

The lookup layer does not provide consistent views of the set of responsible nodes for a given id, i.e., different and concurrent lookups may produce different “correct” results. This makes the task of building a DHT that provides strong semantics complex. The initial work in peer-to-peer DHTs [25, 86] did not provide any guarantees in terms of the recency of the data retrieved from the system, or even of the success of that operation.

Our approach of building a membership service that produces system configura-

tions that guide the system through periods of stability, instead of using a less stable lookup layer that provides inconsistent views of the system membership, allowed us to design simple storage systems that provide strong semantics. Furthermore, we still obtain most of the benefits from the peer-to-peer approach, namely automatic reconfiguration.

Other work tried to improve on this, and proposed DHT designs that provide atomic updates to mutable data [61]. This work assumes that nodes notify other nodes before failing and therefore it does not actually handle failures but only planned node departures.

Another limitation of the lookup layer in peer-to-peer systems is that most of this work does not tolerate Byzantine faults, which makes the outcome of the lookup vulnerable to a single dishonest node in the sequence of nodes that is contacted in the routing process. Castro et al. have proposed extensions to the Pastry peer-to-peer lookup protocol [85] to make it robust against malicious attacks [18]. This prevents an adversary from causing a lookup to return a set of bad nodes as the set of replicas for an item, but this does not solve the basic problem of how to avoid inconsistent views of the system membership.

OceanStore [52, 81] is a DHT that is tolerant of Byzantine faults. OceanStore is a two-tiered system. The primary tier of replicas offers strong consistency for mutable data despite Byzantine faults using the Castro-Liskov BFT state machine replication algorithm [20]. Its design mentions automatic reconfiguration as a goal but does not detail its approach, design, or implementation. Our membership service would be an interesting addition to this system as a means to determine the current membership for the primary tier. The secondary tier is used essentially to propagate self-verifying content with weaker consistency guarantees, so it does not require any trust in the replicas that serve that content. The secondary tier uses peer-to-peer routing to locate replicas for the data.

Another large-scale storage system that tolerates Byzantine faults and is designed with similar goals to peer-to-peer systems is Farsite [4]. Farsite is a file system that uses spare resources from desktop PCs to logically function as a centralized file system.



Farsite does not detail how to detect failed nodes and migrate their responsibilities to new nodes, and it mentions as future work the design of a mechanism to determine which machines to place file replicas on, but no subsequent publications address these issues. Our membership service could be used in Farsite to implement this missing part of its design, although we would have to extend it with measurements of machine availability, as required by their system.

Neither OceanStore nor Farsite addresses the issue of changing the storage protocols (in their case, BFT) to support a dynamic membership. dBFT could be used for this purpose.

## 10.2 Dynamic Replication

To our knowledge we are the first to present a generic methodology for transforming replicated services that work in a static setting into services with the same semantics but support membership changes.

The two example services we implemented, dBQS and dBFT, are related to previous systems.

### 10.2.1 Systems Related to dBQS

dBQS is related to a large body of research in quorum systems. Quorum systems—collections of sets where every two sets have a non-empty intersection—have been used for providing consistent data in distributed settings since the seminal work of Gifford [36] and Thomas [92].

The initial work on quorum systems assumed a static processor universe. More recently, these algorithms have been extended to work in long-lived systems where the processors may dynamically join and leave the system, and the quorums systems are allowed to reconfigure [27, 28, 30, 38, 43, 47, 62, 63].

The aforementioned work assumes benign failures. Quorum systems that tolerate Byzantine failures were initially proposed by Malkhi and Reiter [65]. Our algorithms are more closely related to the one proposed in Phalanx [66], which provides atomic

semantics, but also assumed a static processor universe. Alvisi et al. [6] looked at dynamic Byzantine quorum systems. However, this work assumes a fixed replica set, and simply allows the failure threshold to change throughout the execution. We could easily change the failure threshold by having this information be part of the configuration description. Kong et al. [51] improve Alvisi’s protocol to include a special, fault-free node that monitors the set of servers in the system. When faulty servers are identified, they can be removed from the server set, which reduces the load in the system. However, there is no provision for adding new nodes to the system, and thus the system does not allow the replacement of faulty nodes with new, non-failed ones.

Recently, the work of Martin et al. [67] addresses the same problem that dBQS does of providing reconfigurable Byzantine quorums. Their solution solves the problem of slow clients accessing old replica groups by having their equivalent to our membership service issue new keys to all replicas after each reconfiguration. Old replicas discard the previous keys, and thereafter cannot handle client requests. Our work differs in several respects. First, we designed our system to work at a large scale with many replica groups. Their solution does not scale well with the number of groups in the system because disseminating new keys after each reconfiguration is costly and leads to large configuration descriptions. Second, we designed the membership service in such a way that it can be (and has been) implemented as a Byzantine-fault-tolerant state machine replication group. Their solution does not allow such replication of their equivalent of the membership service because it knows information that cannot be disclosed to any malicious party (it knows the private keys of the replicas that are serving the data). Third, there are problems we dealt with that only arise when there are multiple replica groups, such as the garbage collection of old data. Finally, we implemented of our solution and show performance results.

## 10.2.2 Systems Related to dBFT

dBFT is a form of state machine replication [53, 89]. Previous systems that provide state machine replication and tolerate Byzantine faults are Rampart [79], Se-

cureRing [50], and BFT [20].

Our work is most closely related to BFT, since this is the algorithm we modified. BFT assumes a static processor set and we improve on this work by allowing the replica set to change throughout the execution of the system.

Rampart and SecureRing take a different approach to implementing state machine replication. Both Rampart and SecureRing are based on group communication systems. The idea is to use a feature of the group communication system that implements a reliable, atomic multicast. This is used by servers that, on behalf of clients, multicast client requests to all current group members. Then, all that is required is that all servers keep a copy of the state machine and execute client requests as they are delivered to them by the group communication system.

Using a group communication system has the advantage that it inherits the re-configuration properties of that system: it is possible to add and remove servers from the system, and the multicast of the messages will adapt to it, delivering the client requests only to the current group members. To implement state machine replication, there is still the need for a state transfer protocol for joining (or re-joining) nodes, and some mechanism to authenticate the current group for joining or slow nodes. This protocol is not defined in the aforementioned systems.

The use of group communication systems has the problems mentioned above: poor scalability, and they require that faulty nodes are evicted from the system to make progress, which leads to the possibility of nodes being mistakenly marked as faulty.

### **10.3 Analysis of Membership Dynamics**

Our analysis of the cost of redundancy maintenance in dynamic system was inspired by several proposals of cooperative storage systems. The idea behind a cooperative storage system is to harness the spare storage and spare bandwidth of volunteers to build a storage system simultaneously accessible by many clients.

Some of these systems have been presented above. For instance, Farsite and xFS are based on the idea of using spare resources from desktop PCs in a large corporation,

and peer-to-peer storage systems (e.g., CFS [25], PAST [86], TotalRecall [13]) that try to harness the spare resources of any node connected to the Internet that may wish to participate in the system.

The main contribution here is to point out an important limitation that these systems may face: Their approach only works if the dynamics are limited. We were the first to point out this problem and to present this kind of analysis.

# Chapter 11

## Conclusion

This chapter presents a summary of the main results in the thesis and directions for future work.

### 11.1 Summary

The growing reliance of our society on Internet services demands highly-available systems that provide correct service without interruptions. This thesis describes new techniques for building large-scale, replicated services that tolerate Byzantine faults.

The first part of the thesis described a membership service that supports this technique. The membership service provides servers and clients in the system with a sequence of consistent views of the system membership. Our solution uses several techniques that allows it to scale to large systems, e.g., with tens or hundreds of thousands of servers: we only execute agreement protocols (to decide on membership changes) among a small committee formed by a subset of the members; the system moves in a succession of time intervals called epochs, and we batch all membership changes at the end of an epoch; and we employ a strategy of having a delayed response to failures which allows us to probe system members slowly. The correctness of the membership service depends on realistic and lenient conditions. We only require that no more than one third of the servers in the committee are faulty, and we tolerate arbitrary failures. Furthermore we choose committee members from the current set

of available servers, which means we have a large pool of available servers, and we move the committee around periodically, to avoid having it be a target of attack.

The second part of the thesis presents a generic methodology for transforming large-scale replicated services that assume a fixed membership into services with the same semantics that support a dynamic system membership. The methodology uses the output from the membership service to decide when to reconfigure, and to bring slow nodes up-to-date by presenting them with signed certificates produced by the membership service. The replicated services built using this methodology have reasonable correctness conditions that are not stringent: we only require that each replica group in the system contain less than maximum number of faults the algorithm can tolerate while that group is needed for serving data to clients or for the purposes of state transfer.

We back up our methodology with two example replicated services: dBQS is a system based on Byzantine quorum replication, and dBFT is based on Byzantine-fault-tolerant state machine replication. In both cases we extended existing algorithms that were designed for a static membership to work with a dynamic replica set.

The final contribution of the thesis was to develop a model for the bandwidth cost of maintaining data redundancy in a dynamic system where servers join and leave continuously. We used measured values from different real-world applications to determine values for the parameters of the model, with emphasis on the membership dynamics. This allowed us to make an important conclusion that spare bandwidth, and not spare storage, is likely to be the limiting factor for the deployment of systems that store a large service state and support a highly dynamic membership.

We examined some bandwidth optimization strategies like delaying the response to unreachability to distinguish temporary disconnections from permanent departures, or using erasure codes instead of replication. We found that a delayed response to unreachability leads to significant bandwidth savings. For the use of erasure coding, the gains are dependent on the deployment of the service. For stable deployments, their gains are limited. For unstable, dynamic deployments, there are some bandwidth savings, but not enough to make the system practical: the cost of redundancy

maintenance is still too high.

We implemented the membership service and the two replicated services and our performance evaluation concluded that the systems are practical. Our membership service ran on a large-scale, wide-area testbed and the time the system took to reconfigure is small enough that the system can support relatively frequent reconfigurations (e.g., on the order of few minutes between reconfigurations). Furthermore, we found that the performance of our replicated services when there are changes to the system membership is close to the performance of the services in the static case. And we also determined that the measured impact of superimposing the membership service on instances of the replicated services is small.

## 11.2 Future Work

We divide our future work directions according to the different contributions of the thesis.

### 11.2.1 Membership Service

It would be interesting to use committees for extending the scalability of the membership service. We believe that this is a simple design extension, although the exact details still need to be worked out.

An obstacle to the design of our membership service was the scalability of the APSS protocol with the maximum number of faults tolerated,  $f$ . Our implementation of this protocol did not perform well with  $f = 2$  and was completely impractical for  $f > 2$ . An interesting research area is to investigate new protocols for proactive secret sharing and proactive threshold signatures that work correctly in asynchronous systems and perform well with values of  $f > 1$ .

We believe the membership service could be valuable for applications other than replicated services. For instance, applications like multicast could also be built based on our membership service. The new challenge in this application is that it may be desirable to have a quicker response to unreachability: multicast trees need to be

rearranged quickly when an interior node fails. One option is to push the limits of the system, by having very small epochs, but this may not be enough. Therefore the multicast system may need some form of redundancy to deal with this unavailability.

### 11.2.2 dBQS

Our current design for dBQS is vulnerable to an attack that tries to exhaust the storage at server by creating a large number of objects (of either type). To solve this we need an admission control and quota scheme that limits the number of objects that may exist in the system. For content-hash objects, we could additionally implement a garbage collection scheme that reclaims these objects automatically if they are not referenced from other objects.

Another problem is the fact that the algorithms used in dBQS are vulnerable to Byzantine-faulty clients that can cause violations of atomicity or exhaust the timestamp space (as described in Section 5.4). An interesting research area is to extend quorum protocols that tolerate Byzantine-faulty clients [66] in two ways. First, so that they work with a dynamic membership, similarly to what we have done with the current protocols. Second, so that they tolerate a client that tries to exhaust the timestamp space.

Finally, we could make our transformation more generic, so that we define how an individual phase can be transformed, and then reuse this transformation for different quorum-based protocols, independently of the number of phases or the information collected in each phase. This is similar to what has been proposed in [67].

### 11.2.3 dBFT

It would be interesting to extend dBFT to support multi-item operations. This means that we need to define a global order for operations across different state machines implemented by different replica groups, similarly to what happens in distributed transactions [39]. To implement operations that use more than one object from multiple groups we need some form of two-phase commit protocol [31] between replica



groups to ensure agreement on the values of the serialization points.

It would be interesting to implement more services based on dBFT, and we also want to exploit different designs for our file system implementation. The current design stores an entire file system tree as a single item. This leads to a simpler design for the file system code but also leads to worse load-balancing, since all files and directories in each file system are the responsibility of a single replica group. In the future we could explore more fine-grained partitioning of the service where different files (or even different pages within each file) and directories are different items replicated at different groups.

#### 11.2.4 Analysis of Dynamic Replicated Systems

It would be interesting to validate the analysis presented in Chapter 8 with simulations, using as input the traces of session dynamics for the three deployments we studied. Simulations would help us validate our analytic results, and also would help us study the worst-case behavior of the system.

Our analysis points out an important limitation of peer-to-peer storage systems, namely that volunteer-based systems may not withstand the bandwidth requirements to maintain data redundancy in the presence of a dynamic membership. A promising research direction is to redesign current peer-to-peer storage systems in order to minimize these problems.

One possible scheme is to use surrogates to avoid data movement when nodes join the system. The idea is that when nodes join they become responsible for all items in a certain interval in the id space, but the system may already contain enough replicas for each of these items. To avoid the data transfer associated with joins, we can create surrogates in joining nodes that point to the existing replicas of the data. The details of the design are left as future work.

Another interesting idea for minimizing the bandwidth cost came from TotalRecall [13], a system that delays creating new copies of data until replication levels fall below certain levels. This leads to biasing the storage toward more stable nodes that remain in the system for longer, therefore breaking one of the assumptions behind

our analysis.

In the future, we would like to incorporate these two optimizations in our analysis and simulations, and determine if these are possible solutions for the problem we pointed out.

Finally, a limitation of our analysis is that it assumes that the data is immutable, and that the replicas that are available behave correctly. It would be interesting to extend this analysis for mutable data, and for different assumptions about the behavior of faulty nodes (failstop and Byzantine failures). Note that the analysis for erasure coded redundancy already assumes that we need a fraction of the servers to be available in order to retrieve the data. We believe that the analysis for algorithms like BFT would be similar.

Another interesting point that needs further studying is how to integrate the Byzantine failure assumption with erasure coded redundancy. The current designs that use erasure coded redundancy assume that fragments that are downloaded are always correct. We would like to understand how to design a storage system that uses erasure-coded storage and tolerates Byzantine faults.

# Appendix A

## The MS Modules

In the methodology described in Chapter 4, several tasks, such as checking leases or obtaining missing configurations when nodes skip an epoch, have to be performed by any replicated service. Furthermore, the methodology depends on the fact that servers in the system are notified of new configurations by a membership service.

We encapsulate the service-independent functionality and the implementation of the membership service described in Chapter 3 into separate MS modules that run alongside with the application. There are separate MS modules for clients and servers.

The MS *client* module tries to ensure that the client holds a valid lease all the time. Before a client lease expires, the MS client module tries to renew it, so that clients will hold valid leases whenever possible.

After a client receives a reply that belongs to an epoch  $e$ , the client can invoke a `check-lease` function that verifies if the client is holding a valid lease for  $e$ .

In the process of renewing a lease, the client MS module may find out about a new configuration, e.g., the old MS can return the new configuration delta to the client. In this case the client MS module notifies the client via a `new-config` upcall. This upcall passes the delta to the previous configuration and the signature that authenticates the new configuration as arguments.

The MS client module also provides a `move-config` function. This is called by the client proxy for the replicated service when it learns about the a new configuration as a result of a reply from a server. (As mentioned earlier, this reply contains a

certificate for the new configuration.)

The MS *server* module implements the MS functionality. It acts as an MS replica whenever necessary, issuing probes, executing the MS service, producing new configurations, etc. If the server is not an MS replica it still needs to handle probes, handle lease aggregation request for clients, and multicast new configurations.

When the server module discovers a new configuration it immediately notifies the replicated service via a **new-config** upcall. This upcall passes the delta to the previous configuration and the signature that authenticates the new configuration as arguments.

When the replicated service receives a message that contains a new configuration that configuration can also be passed directly to the MS module via a **move-config** call. This call takes as an argument the new configuration delta and the signature of that configuration, and returns an error if the signature is not valid.

A server may pass a configuration delta that is more than one epoch ahead the current epoch held by the server. In this case, it is impossible to reconstruct the current configuration from the delta, since the missing intermediate configuration are required to reconstruct the system membership. In this case, the **move-config** call will act as a hint to the MS module that the node skipped some epochs. The MS module will try to obtain the missing epochs, contacting the MS replicas if needed. As the MS module obtains the missing epochs, it will invoke the **move-config** upcall on the replicated service.

# Appendix B

## I/O Automata Code for dBQS

This appendix presents the client, replica, and membership service I/O automata code. Before we present this, we will make some clarifications about the notation in this description.

The notation used in our IOA specifications follows closely the one in [60]. We omit channel automata, but we assume these are lossy reordering channels as defined in [60]. We use the following notation for cryptographic functions.  $\langle m \rangle_{\sigma_n}$  means that node  $n$  digitally signs  $m$  and appends the signature to the message.  $p.verify(x, \sigma)$  is a boolean function that returns true if the public key  $p$  verifies that  $\sigma$  is a correct signature for the content  $x$ . We denote  $p_c$  as the well know client public key, corresponding to the private key used by authorized writers to sign the data and its associated timestamp (this signature is included in the data block's header). We assume the existence of a  $random()$  function that produces a random value (for nonces) with probability of collision equal to 0. We simplify the notation of picking a timestamp greater than  $t$  by writing  $t + 1$ . In practice timestamps are  $\langle \text{counter}, \text{client id} \rangle$  pairs, so this corresponds to incrementing the counter and appending its own client id. For clarity, in this presentation we will separate the set of process identifiers into a set of server identifiers  $S$ , and client identifiers  $C$ .

## Client Automaton

### Signature:

Input:  $\text{READ}_c$   
 $\text{WRITE}(v)_c, v \in V$   
 $\text{RECEIVE}(\langle \text{READ-REPLY}, val, ts, \sigma, nonce \rangle_{\sigma_i})_{i,c}, val \in V, ts \in \mathcal{T}, \sigma \in \Sigma, nonce \in \mathcal{N}$   
 $\text{RECEIVE}(\langle \text{READ-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle)_{i,c}$   
 $\text{RECEIVE}(\langle \text{READ-REPLY}, \text{ERR\_UPGRADE\_CONFIG}, next\_epoch, next\_config, \sigma_{ms} \rangle)_{i,c}, next\_epoch \in \mathbb{N},$   
 $next\_config \in \mathcal{G}, \sigma_{ms} \in \Sigma$   
 $\text{RECEIVE}(\langle \text{WRITE-REPLY}, \text{ACK}, nonce, t \rangle_{\sigma_i})_{i,c}, nonce \in \mathcal{N}, t \in \mathcal{T}$   
 $\text{RECEIVE}(\langle \text{WRITE-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle)_{i,c}$   
 $\text{RECEIVE}(\langle \text{WRITE-REPLY}, \text{ERR\_UPGRADE\_CONFIG}, next\_epoch, next\_config, \sigma_{ms} \rangle)_{i,c}, next\_epoch \in \mathbb{N},$   
 $next\_config \in \mathcal{G}, \sigma_{ms} \in \Sigma$   
 $\text{RECEIVE}(\langle \text{POLL-CONFIG-REPLY}, nonce, epoch, config, config\_signature \rangle)_{ms,c}, nonce \in \mathcal{N}, epoch \in \mathbb{N},$   
 $config \in \mathcal{G}, config\_signature \in \Sigma$   
Internal:  $\text{POLL-CFG}_c$   
Output:  $\text{READ-ACK}(v)_c, v \in V$   
 $\text{WRITE-ACK}_c$   
 $\text{SEND}(m)_{c,i}$

### State:

$epoch_c \in \mathbb{N}$ , initially 0  
 $config_c \in \mathcal{G}$ , initially  $c_0$   
 $config\_signature_c \in \Sigma$ , initially  $\sigma_0$   
 $previous\_config_c \in \mathcal{G}$ , initially  $\{ \}$   
 $status_c \in \{idle, read\_p1, read\_p2, read\_done, write\_p1, write\_p2, write\_done\}$ , initially  $idle$   
 $nonce_c \in \mathcal{N}$ , initially 0  
 $max\_ts_c \in \mathcal{T}$ , initially  $t_0, t_0 < t, \forall t \in \mathcal{T}$   
 $max\_val_c \in V$ , initially  $v_0$   
 $max\_sig_c \in \Sigma$ , initially  $\sigma_0$   
 $reply\_set \in 2^S$ , initially  $\{ \}$   
 $reply\_set\_sec\_phase \in 2^S$ , initially  $\{ \}$   
 $num\_agreeing\_replies_c \in \mathbb{N}$ , initially 0  
 $val\_to\_write_c \in V$ , initially  $v_0$   
 $sig\_to\_write_c \in \Sigma$ , initially  $\sigma_0$   
 $poll\_nonce_c \in \mathcal{N}$ , initially 0  
for every  $j \in S$ :  
 $send\_buffer(j)_c$ , a FIFO queue of messages, initially empty

### Transitions:

READ<sub>c</sub>

Eff:  $status_c := read\_p1$   
 $nonce_c := random()$   
 $max\_ts_c := t_0$   
 $reply\_set_c := \{\}$   
 $num\_agreeing\_replies_c := 0$   
for all  $j \in \{j \in S : \langle j, p \rangle \in config_c\}$   
    add  $\langle READ, epoch_c, nonce \rangle$  to  $send\_buffer(j)_c$

WRITE( $v$ )<sub>c</sub>

Eff:  $status_c := write\_p1$   
 $nonce_c := random()$   
 $max\_ts_c := t_0$   
 $reply\_set_c := \{\}$   
 $val\_to\_write_c := v$   
for all  $j \in \{j \in S : \langle j, p \rangle \in config_c\}$   
    add  $\langle WRITE, epoch_c, nonce \rangle$  to  $send\_buffer(j)_c$

RECEIVE( $\langle READ-REPLY, val, ts, \sigma, nonce \rangle_{\sigma_i}$ )<sub>i,c</sub>

Eff: if  $\exists p : \langle i, p \rangle \in config_c$  and  $i \notin reply\_set_c$  and  $nonce_c = nonce$  and  $p.verify(nonce, ts, \sigma_i)$  and  $p_c.verify(val, ts, \sigma)$  then  
    if  $status_c = read\_p1$  or  $status_c = read\_p2$  then  
        if  $ts = max\_ts_c$  then  
             $num\_agreeing\_replies_c := num\_agreeing\_replies_c + 1$   
        if  $ts > max\_ts_c$  then  
             $max\_ts_c := ts$   
             $max\_val_c := val$   
             $max\_sig_c := \sigma$   
             $num\_agreeing\_replies_c := 1$   
             $reply\_set_c := reply\_set_c \cup \{i\}$   
            if  $num\_agreeing\_replies_c = 2f + 1$  then  
                 $status_c := read\_done$   
            else if  $|reply\_set_c| = 2f + 1$   
                 $status_c := read\_p2$   
                 $reply\_set\_sec\_phase_c := \{\}$   
                for all  $j \in \{j \in S : \langle j, p \rangle \in config_c\}$   
                    add  $\langle WRITE, epoch_c, max\_ts_c, max\_val_c, max\_sig_c, nonce \rangle$  to  $send\_buffer(j)_c$   
    if  $status_c = write\_p1$  then  
        if  $ts > max\_ts_c$  then  
             $max\_ts_c := ts$   
         $reply\_set_c := reply\_set_c \cup \{i\}$   
        if  $|reply\_set_c| = 2f + 1$  then  
             $status_c := write\_p2$   
             $reply\_set\_sec\_phase_c := \{\}$   
             $sig\_to\_write_c := p_c.sign(val\_to\_write_c, max\_ts_c + 1)$   
            for all  $j \in \{j \in S : \langle j, p \rangle \in config_c\}$   
                add  $\langle WRITE, epoch_c, max\_ts_c + 1, val\_to\_write_c, sig\_to\_write_c, nonce \rangle$  to  $send\_buffer(j)_c$

RECEIVE( $\langle \text{READ-REPLY, ERR\_NEED\_CONFIG} \rangle_{i,c}$ )

Eff: if  $\exists p : \langle i, p \rangle \in \text{config}_c$  and  $(\text{status}_c = \text{read\_p1}$  or  $\text{status}_c = \text{write\_p1})$  then  
 add  $\langle \text{NEW-CONFIG, epoch}_c, \text{config}_c, \text{config\_signature}_c \rangle$  to  $\text{send\_buffer}(i)_c$   
 add  $\langle \text{READ, epoch}_c, \text{nonce} \rangle$  to  $\text{send\_buffer}(i)_c$

RECEIVE( $\langle \text{READ-REPLY, ERR\_UPGRADE\_CONFIG, next\_epoch, next\_config, } \sigma_{ms} \rangle_{i,c}$ )

Eff: if  $\text{next\_epoch} > \text{epoch}_c$  and  $p_{ms}.\text{verify}(\text{next\_epoch}, \text{next\_config}, \sigma_{ms})$  then  
 $\text{epoch}_c := \text{next\_epoch}$   
 $\text{previous\_config}_c := \text{config}_c$   
 $\text{config}_c := \text{next\_config}$   
 $\text{config\_signature}_c := \sigma_{ms}$   
 if  $\text{status}_c = \text{read\_p1}$  or  $\text{status}_c = \text{write\_p1}$  then  
 $\text{nonce}_c := \text{random}()$   
 $\text{max\_ts}_c := t_0$   
 $\text{reply\_set}_c := \{\}$   
 $\text{num\_agreeing\_replies}_c := 0$   
 for all  $j \in \{j \in S : \langle j, p \rangle \in \text{config}_c\}$   
 add  $\langle \text{READ, epoch}_c, \text{nonce} \rangle$  to  $\text{send\_buffer}(j)_c$   
 if  $\text{status} = \text{read\_p2}$  then  
 $\text{nonce}_c := \text{random}()$   
 $\text{reply\_set}_c := \{\}$   
 $\text{num\_agreeing\_replies}_c := 0$   
 $\text{reply\_set\_sec\_phase}_c := \{\}$   
 for all  $j \in \{j \in S : \langle j, p \rangle \in \text{config}_c\}$   
 add  $\langle \text{WRITE, epoch}_c, \text{max\_ts}_c, \text{max\_val}_c, \text{max\_sig}_c, \text{nonce} \rangle$  to  $\text{send\_buffer}(j)_c$   
 if  $\text{status} = \text{write\_p2}$  then  
 $\text{nonce}_c := \text{random}()$   
 $\text{reply\_set}_c := \{\}$   
 $\text{num\_agreeing\_replies}_c := 0$   
 $\text{reply\_set\_sec\_phase}_c := \{\}$   
 for all  $j \in \{j \in S : \langle j, p \rangle \in \text{config}_c\}$   
 add  $\langle \text{WRITE, epoch}_c, \text{max\_ts}_c + 1, \text{val\_to\_write}_c, \text{sig\_to\_write}_c, \text{nonce} \rangle$  to  $\text{send\_buffer}(j)_c$

RECEIVE( $\langle \text{WRITE-REPLY, ACK, nonce, } t \rangle_{\sigma_i} \rangle_{i,c}$ )

Eff: if  $\exists p : \langle i, p \rangle \in \text{config}_c$  and  $i \notin \text{reply\_set\_sec\_phase}_c$  and  $((\text{status}_c = \text{read\_p2}$  and  $t = \text{max\_ts}_c)$  or  
 $(\text{status}_c = \text{write\_p2}$  and  $t = \text{max\_ts}_c + 1))$  and  $\text{nonce}_c = \text{nonce}$  and  $p.\text{verify}(\text{nonce}, t, \sigma_i)$  then  
 $\text{reply\_set\_sec\_phase}_c := \text{reply\_set\_sec\_phase}_c \cup \{i\}$   
 if  $|\text{reply\_set\_sec\_phase}_c| = 2f + 1$  then  
 if  $\text{status}_c = \text{read\_p2}$  then  
 $\text{status}_c := \text{read\_done}$   
 if  $\text{status}_c = \text{write\_p2}$  then  
 $\text{status}_c := \text{write\_done}$

RECEIVE( $\langle \text{WRITE-REPLY, ERR\_NEED\_CONFIG} \rangle_{i,c}, \text{nonce} \in \mathcal{N}$ )

Eff: if  $\exists p : \langle i, p \rangle \in \text{config}_c$  and  $(\text{status}_c = \text{read\_p2}$  or  $\text{status}_c = \text{write\_p2})$  then  
 add  $\langle \text{NEW-CONFIG, epoch}_c, \text{config}_c, \text{config\_signature}_c \rangle$  to  $\text{send\_buffer}(i)_c$   
 if  $\text{status} = \text{read\_p2}$  then  
 add  $\langle \text{WRITE, epoch}_c, \text{max\_ts}_c, \text{max\_val}_c, \text{max\_sig}_c, \text{nonce} \rangle$  to  $\text{send\_buffer}(i)_c$   
 if  $\text{status} = \text{write\_p2}$  then  
 add  $\langle \text{WRITE, epoch}_c, \text{max\_ts}_c + 1, \text{val\_to\_write}_c, \text{sig\_to\_write}_c, \text{nonce} \rangle$  to  $\text{send\_buffer}(i)_c$



RECEIVE( $\langle$ WRITE-REPLY, ERR\_UPGRADE\_CONFIG,  $next\_epoch$ ,  $next\_config$ ,  $\sigma_{ms}$  $\rangle$ ) $_{i,c}$   
 Eff: if  $next\_epoch > epoch_c$  and  $p_{ms}.verify(next\_epoch, next\_config, \sigma_{ms})$  then  
    $epoch_c := next\_epoch$   
    $previous\_config_c := config_c$   
    $config_c := next\_config$   
    $config\_signature_c := \sigma_{ms}$   
   if  $status_c = read\_p1$  or  $status_c = write\_p1$  then  
      $nonce_c := random()$   
      $max\_ts_c := t_0$   
      $reply\_set_c := \{\}$   
      $num\_agreeing\_replies_c := 0$   
     for all  $j \in \{j \in S : \langle j, p \rangle \in config_c\}$   
       add  $\langle READ, epoch_c, nonce \rangle$  to  $send\_buffer(j)_c$   
   if  $status = read\_p2$  then  
      $nonce_c := random()$   
      $reply\_set_c := \{\}$   
      $num\_agreeing\_replies_c := 0$   
      $reply\_set\_sec\_phase_c := \{\}$   
     for all  $j \in \{j \in S : \langle j, p \rangle \in config_c\}$   
       add  $\langle WRITE, epoch_c, max\_ts_c, max\_val_c, max\_sig_c, nonce \rangle$  to  $send\_buffer(j)_c$   
   if  $status = write\_p2$  then  
      $nonce_c := random()$   
      $reply\_set_c := \{\}$   
      $num\_agreeing\_replies_c := 0$   
      $reply\_set\_sec\_phase_c := \{\}$   
     for all  $j \in \{j \in S : \langle j, p \rangle \in config_c\}$   
       add  $\langle WRITE, epoch_c, max\_ts_c + 1, val\_to\_write_c, sig\_to\_write_c, nonce \rangle$  to  $send\_buffer(j)_c$

SEND( $m$ ) $_{c,i}$   
 Pre:  $m$  is first on  $send\_buffer(i)_c$   
 Eff: remove first element of  $send\_buffer(i)_c$

READ-ACK( $v$ ) $_c$   
 Pre:  $status_c = read\_done$   
    $v = max\_val_c$   
 Eff:  $status_c = idle$

WRITE-ACK $_c$   
 Pre:  $status_c = write\_done$   
 Eff:  $status_c = idle$

POLL-CFG $_c$   
 Pre: none  
 Eff:  $poll\_nonce_c := random()$   
   add  $\langle POLL\_CONFIG, poll\_nonce_c \rangle$  to  $send\_buffer(ms)_c$

RECEIVE( $\langle$ POLL-CONFIG-REPLY,  $nonce$ ,  $epoch$ ,  $config$ ,  $config\_signature$  $\rangle$ ) $_{ms,c}$   
 Eff: if  $nonce = poll\_nonce_c$  and  $epoch > epoch_c$  and  $verify(epoch, config, config\_signature)$  and  
    $p_{ms}.verify(epoch, nonce, \sigma_{ms})$  then  
    $epoch_c := epoch$   
    $config_c := config$   
    $config\_signature_c := config\_signature$   
    $poll\_nonce_c := 0$

## Replica Automaton

### Signature:

Input:           RECEIVE( $\langle \text{READ}, epoch, nonce \rangle_{c,i}$ )  
                   RECEIVE( $\langle \text{WRITE}, epoch, ts, val, \sigma, nonce \rangle_{c,i}$ )  
                   RECEIVE( $\langle \text{NEW-CONFIG}, epoch, config, \sigma_{ms} \rangle_{n,i}$ ),  $epoch \in \mathbb{N}, config \in \mathcal{G}, \sigma_{ms} \in \Sigma$   
                   RECEIVE( $\langle \text{STATE-TRF-READ}, epoch, nonce \rangle_{j,i}$ )  
                   RECEIVE( $\langle \text{STATE-TRF-READ-REPLY}, val, ts, \sigma, nonce \rangle_{\sigma_i} \rangle_{j,i}$ ),  $val \in V, ts \in \mathcal{T}, \sigma \in \Sigma, nonce \in \mathcal{N}$   
                   RECEIVE( $\langle \text{STATE-TRF-READ-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle_{j,i}$ )  
                   RECEIVE( $\langle \text{POLL-EPOCH}, nonce \rangle_{ms,i}$ ),  $nonce \in \mathcal{N}$   
                   RECEIVE( $\langle \text{CFG-QUERY-REPLY}, e, nonce, cfg, \sigma_{cfg} \rangle_{\sigma_{ms}} \rangle_{ms,i}$ ),  $e \in \mathbb{N}, nonce \in \mathcal{N}, \sigma_{cfg} \in \Sigma$   
                   RECEIVE( $\langle \text{CFG-QUERY-REPLY}, e, nonce, \perp \rangle_{\sigma_{ms}} \rangle_{ms,i}$ ),  $e \in \mathbb{N}, nonce \in \mathcal{N} \in \Sigma$   
                   FAIL<sub>*i*</sub>

Output:           SEND( $m$ )<sub>*i,n*</sub>

Internal:         GOSSIP-CONFIG<sub>*i*</sub>

### State:

$val_i \in V$ , initially  $v_0$   
 $ts_i \in \mathcal{T}$ , initially  $t_0$ ,  $t_0 < t, \forall t \in \mathcal{T}$   
 $sig_i \in \Sigma$ , initially  $\sigma_{v_0, t_0}$   
 $epoch_i \in \mathbb{N}$ , initially 0  
 $config_i \in \mathcal{G}$ , initially  $c_0$   
 $config\_signature_i \in \Sigma$ , initially  $\sigma_0$   
 $previous\_config_i \in \mathcal{G}$ , initially  $\{ \}$   
 $nonce_i \in \mathcal{N}$ , initially 0  
 $in\_state\_trf_i \in \text{boolean}$ , initially *false*  
 $reply\_set_i \in 2^S$ , initially  $\{ \}$   
 $epoch\_goal_i \in \mathbb{N}$ , initially 0  
 $in\_cfg\_poll_i \in \text{boolean}$ , initially *false*  
 for every  $n \in S \cup C$ :  
    $send\_buffer(n)_i$ , a FIFO queue of messages, initially empty  
    $pending\_reads_i$ , a FIFO queue of elements in  $C \times \mathcal{N}$ , initially empty  
    $pending\_st\_reads_i$ , a FIFO queue of elements in  $S \times \mathcal{N}$ , initially empty  
    $faulty_i \in \text{boolean}$ , initially *false*

### Transitions:

RECEIVE( $\langle \text{READ}, epoch, nonce \rangle_{c,i}$ )  
 Eff: if  $epoch = epoch_i$  then  
   if  $\neg in\_state\_trf_i$  then  
     add  $\langle \text{READ-REPLY}, val_i, ts_i, sig_i, nonce \rangle_{\sigma_i}$  to  $send\_buffer(c)_i$   
   else  
     add  $\langle c, nonce \rangle$  to  $pending\_reads_i$   
 if  $epoch > epoch_i$  then  
   add  $\langle \text{READ-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle$  to  $send\_buffer(c)_i$   
 if  $epoch < epoch_i$  then  
   add  $\langle \text{READ-REPLY}, \text{ERR\_UPGRADE\_CONFIG}, epoch_i, config_i, config\_signature_i \rangle$  to  $send\_buffer(c)_i$

RECEIVE( $\langle \text{WRITE}, epoch, ts, val, \sigma, nonce \rangle_{c,i}$ )

Eff: if  $p_c.verify(val, ts, \sigma)$  then  
  if  $epoch = epoch_i$  then  
    add  $\langle \text{WRITE-REPLY}, \text{ACK}, nonce, ts \rangle_{\sigma_i}$  to  $send\_buffer(c)_i$   
    if  $ts > ts_i$  then  
       $val_i := val$   
       $ts_i := ts$   
       $sig_i := \sigma$   
  if  $epoch > epoch_i$  then  
    add  $\langle \text{READ-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle$  to  $send\_buffer(c)_i$   
  if  $epoch < epoch_i$  then  
    add  $\langle \text{READ-REPLY}, \text{ERR\_UPGRADE\_CONFIG}, epoch_i, config_i, config\_signature_i \rangle$  to  $send\_buffer(c)_i$

RECEIVE( $\langle \text{NEW-CONFIG}, epoch, config, \sigma_{ms} \rangle_{n,i}$ )

Eff: if  $p_{ms.verify}(epoch, config, \sigma_{ms})$  and  $epoch > epoch_i$  then  
  if  $\neg in\_state\_trf_i$  then  
    if  $epoch = epoch_i + 1$  then  
       $epoch_i := epoch$   
       $previous\_config_i := config$   
       $config_i := config$   
       $config\_signature_i := \sigma_{ms}$   
      if  $i \in config_i$  and  $i \notin previous\_config_i$  then  
         $in\_state\_trf_i := true$   
         $epoch\_goal := epoch$   
         $reply\_set_i := \{\}$   
         $nonce_i := random()$   
        for all  $j \in \{j \in S : \langle j, p \rangle \in previous\_config_i\}$   
          add  $\langle \text{STATE-TRF-READ}, epoch_i, nonce_i \rangle$  to  $send\_buffer(j)_i$   
    else  
       $in\_cfg\_poll_i := true$   
       $epoch\_goal_i := epoch$   
       $nonce_i := random()$   
      add  $\langle \text{CFG-QUERY}, epoch_i + 1, nonce_i \rangle$  to  $send\_buffer(ms)_i$   
  else  
    if  $epoch > epoch\_goal_i$   
       $epoch\_goal_i := epoch$

RECEIVE( $\langle \text{POLL-EPOCH}, nonce \rangle_{ms,i}$ )

Eff: if  $in\_state\_trf_i$  then

add  $\langle \text{POLL-EPOCH-REPLY}, nonce, epoch_i - 1 \rangle_{\sigma_i}$  to  $send\_buffer(ms)_i$

else

add  $\langle \text{POLL-EPOCH-REPLY}, nonce, epoch_i \rangle_{\sigma_i}$  to  $send\_buffer(ms)_i$

RECEIVE( $\langle \text{STATE-TRF-READ}, epoch, nonce \rangle_{j,i}$ )

Eff: if  $epoch < epoch_i$  then

add  $\langle \text{STATE-TRF-READ-REPLY}, val_i, ts_i, sig_i, nonce \rangle_{\sigma_i}$  to  $send\_buffer(j)_i$

if  $epoch > epoch_i$  then

add  $\langle \text{STATE-TRF-READ-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle$  to  $send\_buffer(j)_i$

if  $epoch = epoch_i$  then

if  $\neg in\_state\_trf_i$  then

add  $\langle \text{STATE-TRF-READ-REPLY}, val_i, ts_i, sig_i, nonce \rangle_{\sigma_i}$  to  $send\_buffer(j)_i$

else

add  $\langle j, nonce \rangle$  to  $pending\_st\_reads_i$

RECEIVE( $\langle \text{STATE-TRF-READ-REPLY}, val, ts, \sigma, nonce \rangle_{\sigma_j}$ ) $_{j,i}$

Eff: if  $in\_state\_trf_i$  and  $\exists p : \langle j, p \rangle \in previous\_config_i$  and  $j \notin reply\_set_i$  and  $nonce_i = nonce$  and  $p.verify(nonce, ts, \sigma_j)$  and  $p.verify(val, ts, \sigma)$  then

if  $ts > ts_i$  then

$val_i := val$

$ts_i := ts$

$sig_i := \sigma$

$reply\_set_i := reply\_set_i \cup \{j\}$

if  $|reply\_set_i| = 2f + 1$  then

$in\_state\_trf_i := false$

for every  $\langle c, n \rangle \in pending\_reads_i$

add  $\langle \text{READ-REPLY}, val_i, ts_i, sig_i, n \rangle_{\sigma_i}$  to  $send\_buffer(c)_i$

remove all elements from  $pending\_reads_i$

for every  $\langle j, n \rangle \in pending\_st\_reads_i$

add  $\langle \text{STATE-TRF-READ-REPLY}, val_i, ts_i, sig_i, n \rangle_{\sigma_i}$  to  $send\_buffer(j)_i$

remove all elements from  $pending\_st\_reads_i$

if  $epoch\_goal_i > epoch_i$  then

$nonce_i := random()$

add  $\langle \text{CFG-QUERY}, epoch_i + 1, nonce_i \rangle$  to  $send\_buffer(ms)_i$

$in\_cfg\_poll_i := true$

RECEIVE( $\langle \text{STATE-TRF-READ-REPLY}, \text{ERR\_NEED\_CONFIG} \rangle_{j,i}$ )

Eff: if  $\exists p : \langle j, p \rangle \in previous\_config_i$  and  $in\_state\_trf_i$  then

add  $\langle \text{NEW-CONFIG}, epoch_i, config_i, config\_signature_i \rangle$  to  $send\_buffer(j)_i$

add  $\langle \text{STATE-TRF-READ}, epoch_i, nonce \rangle$  to  $send\_buffer(j)_i$

RECEIVE( $\langle \text{CFG-QUERY-REPLY}, e, \text{nonce}, \text{cfg}, \sigma_{\text{cfg}} \rangle_{\sigma_{ms}}$ ) $_{ms,i}$   
 Eff: if  $\text{in\_cfg\_poll}_i$  and  $\text{nonce}_i = \text{nonce}$  and  $e = \text{epoch}_i + 1$  and  $p_{ms}.\text{verify}(e, \text{nonce}, \sigma_{ms})$  and  $p_{ms}.\text{verify}(e, \text{cfg}, \sigma_{\text{cfg}})$  then  
    $\text{epoch}_i := e$   
    $\text{previous\_config}_i := \text{config}_i$   
    $\text{config}_i := \text{cfg}$   
    $\text{config\_signature}_i := \sigma_{\text{cfg}}$   
   if  $i \in \text{config}_i$  and  $i \notin \text{previous\_config}_i$  then  
      $\text{in\_cfg\_poll}_i := \text{false}$   
      $\text{in\_state\_trf}_i := \text{true}$   
      $\text{reply\_set}_i := \{\}$   
      $\text{ts}_i := t_0$   
      $\text{nonce}_i := \text{random}()$   
     for all  $j \in \{j \in S : \langle j, p \rangle \in \text{previous\_config}_i\}$   
       add  $\langle \text{STATE-TRF-READ}, \text{epoch}_i, \text{nonce}_i \rangle$  to  $\text{send\_buffer}(j)_i$   
   else  
     if  $\text{epoch\_goal}_i > \text{epoch}_i$  then  
        $\text{nonce}_i := \text{random}()$   
       add  $\langle \text{CFG-QUERY}, \text{epoch}_i + 1, \text{nonce}_i \rangle$  to  $\text{send\_buffer}(ms)_i$   
     else  
        $\text{in\_cfg\_poll}_i := \text{false}$

RECEIVE( $\langle \text{CFG-QUERY-REPLY}, e, \text{nonce}, \perp \rangle_{\sigma_{ms}}$ ) $_{ms,i}$   
 Eff: if  $\text{in\_cfg\_poll}_i$  and  $\text{nonce}_i = \text{nonce}$  and  $e = \text{epoch}_i + 1$  and  $p_{ms}.\text{verify}(e, \text{nonce}, \perp, \sigma_{ms})$  then  
    $\text{ts}_i := t_0$   
   if  $\text{epoch\_goal}_i > \text{epoch}_i$  then  
      $\text{nonce}_i := \text{random}()$   
     add  $\langle \text{CFG-QUERY}, \text{epoch}_i + 1, \text{nonce}_i \rangle$  to  $\text{send\_buffer}(ms)_i$   
   else  
      $\text{in\_cfg\_poll}_i := \text{false}$   
      $\text{config}_i := \{i\}$

FAIL $_i$   
 Eff:  $\text{faulty}_i := \text{true}$

SEND( $m$ ) $_{i,n}$   
 Pre:  $m$  is first on  $\text{send\_buffer}(n)_i$   
 Eff: remove first element of  $\text{send\_buffer}(n)_i$

GOSSIP-CONFIG $_i$   
 Pre: none  
 Eff: add  $\langle \text{NEW-CONFIG}, \text{epoch}_i, \text{config}_i, \text{config\_signature}_i \rangle$  to  $\text{send\_buffer}(j)_i$ ,  $j \in S$

# Membership Service Automaton

## Signature:

Input:           RECEIVE( $\langle \text{POLL-CONFIG}, nonce \rangle_{c,ms}$ ),  $nonce \in \mathcal{N}$   
                   RECEIVE( $\langle \text{POLL-EPOCH-REPLY}, nonce, e \rangle_{\sigma_i}$ ) $_{i,ms}$ ),  $nonce \in \mathcal{N}, e \in \mathbb{N}$   
                   RECEIVE( $\langle \text{CFG-QUERY}, e, nonce \rangle_{i,ms}$ ),  $nonce \in \mathcal{N}, e \in \mathbb{N}$

Output:           SEND( $m$ ) $_{ms,n}$

Internal:        RECONFIGURE( $cfg$ ) $_{ms}$ ,  $cfg \in \mathcal{G}$   
                   POLL-SERVER( $i$ ) $_{ms}$ ,  $i \in S$

## State:

$epoch_{ms} \in \mathbb{N}$ , initially 0  
 $config_{ms} \in \mathcal{G}$ , initially  $c_0$   
 $config\_signature_{ms} \in \Sigma$ , initially  $\sigma_0$   
 $sent\_nonce_{ms} \in S \rightarrow \mathcal{N}$ , initially everywhere 0  
 $config\_history \in \mathbb{N} \rightarrow (\mathcal{G} \times \Sigma) \cup \{\perp, \top\}$ , initially  $config\_history(0) = c_0$ ,  $config\_history(e) = \top$ ,  $e > 0$   
 $transferred \in \mathbb{N} \rightarrow 2^S$ , initially everywhere  $\{ \}$   
 for every  $n \in S \cup C$ :  
    $send\_buffer(n)_{ms}$ , a FIFO queue of messages, initially empty

## Transitions:

RECEIVE( $\langle \text{POLL-CONFIG}, nonce \rangle_{c,ms}$ )  
 Eff: add  $\langle \text{POLL-CONFIG-REPLY}, nonce, epoch_{ms}, config_{ms}, config\_signature_{ms} \rangle_{\sigma_{ms}}$  to  $send\_buffer(c)_{ms}$

RECEIVE( $\langle \text{POLL-EPOCH-REPLY}, nonce, e \rangle_{\sigma_i}$ ) $_{i,ms}$   
 Eff: if  $nonce = sent\_nonce(i)_{ms}$  and  $verify(nonce, e, \sigma_i)$  then  
   for all  $e' \leq e$  such that  $\exists \sigma : \langle i, \sigma \rangle \in config\_history(e')$   
      $transferred(e') := transferred(e') \cup \{i\}$   
   if  $|transferred(e')| = 2f + 1$  then  
      $config\_history(e_{old}) := \perp, \forall e_{old} < e'$

RECEIVE( $\langle \text{CFG-QUERY}, e, nonce \rangle_{i,ms}$ )  
 Eff: if  $config\_history(e)_{ms} = \perp$  then  
   add  $\langle \text{CFG-QUERY-REPLY}, e, nonce, \perp \rangle_{\sigma_{ms}}$  to  $send\_buffer(i)_{ms}$   
   if  $config\_history(e)_{ms} = \langle cfg, \sigma_{cfg} \rangle$  then  
     add  $\langle \text{CFG-QUERY-REPLY}, e, nonce, cfg, \sigma_{cfg} \rangle_{\sigma_{ms}}$  to  $send\_buffer(i)_{ms}$

SEND( $m$ ) $_{i,n}$   
 Pre:  $m$  is first on  $send\_buffer(n)_i$   
 Eff: remove first element of  $send\_buffer(n)_i$

RECONFIGURE( $cfg$ ) $_{ms}$

Pre:  $cfg \in \mathcal{G}$

$|cfg| = 3f + 1$

Eff:  $epoch_{ms} := epoch_{ms} + 1$

$config_{ms} := cfg$

$config\_signature_{ms} := sign(epoch_{ms}, config_{ms})$

$config\_history(epoch_{ms}) := \langle cfg, config\_signature_{ms} \rangle$

for all  $i \in cfg$ :

add  $\langle \text{NEW-CONFIG}, epoch_{ms}, config_{ms}, config\_signature_{ms} \rangle$  to  $send\_buffer(i)_{ms}$

POLL-SERVER( $i$ ) $_{ms}, i \in S$

Pre: none

Eff:  $sent\_nonce(i)_{ms} := random()$

add  $\langle \text{POLL-EPOCH}, sent\_nonce(i)_{ms} \rangle$  to  $send\_buffer(i)_{ms}$





# Appendix C

## Proof of Correctness of the dBQS Algorithms

This appendix discusses the proof of atomic consistency of the algorithms used in dBQS (Chapter 5). To prove this, we only need to show that the timestamps associated with the values induce a partial-order that obeys four properties, which are enumerated in the proof of Theorem 1. This implies the atomicity property, using Lemma 13.16 in [60].

Throughout the proof, we will refer to the correctness condition for dBQS (presented in Chapter 4) as being the conjunction of two conditions (C1 and C2) which correspond to the two sub-conditions that govern the end of the window of vulnerability as presented in Chapter 4.

The proof starts with two definitions. Then we prove a few lemmas that will be helpful in proving Theorem 1, which states that our algorithm implements an atomic variable.

**Definition** We say that the read or a write phase of an operation *completes in epoch*  $e$  if during that phase the client collects valid replies in a quorum for epoch  $e$  before it upgrades to a subsequent epoch (and thus completes the operation or moves on to the next phase).

**Definition** Given a read or write operation  $\pi$ , we define  $epoch(\pi)$  to be the epoch

in which the last phase of the operation completes (we define the last phase to be the read phase of a read operation if that phase gathered enough identical replies to skip the write phase, or otherwise to be the write phase of the operation), and  $timestamp(\pi)$  to be the timestamp associated with the value read in a read operation, or written in a write operation. Similarly, when a replica upgrades a configuration and transfers state from the previous replicas, we define  $timestamp(st_i)$  to be the timestamp associated with the value replica  $i$  read during state transfer (the highest timestamp in the quorum of STATE-TRF-READ-REPLY messages for that epoch).

**Lemma 1** If there exists an operation  $\pi$  (a read or a write) such that  $epoch(\pi) = e$  and  $timestamp(\pi) = t$ , then for any replica  $i$  that transfers state from epoch  $e$  to epoch  $e + 1$ ,  $timestamp(st_i) \geq t$  (i.e., state transfer will read a timestamp that is greater or equal than  $t$ ).

**Proof.** Consider the quorum where the last phase completes and an arbitrary read quorum that is used for state transfer. Those quorums intersect in at least one nonfaulty replica (given correctness condition C1, the state transfer operation will collect replies from at most  $f$  faulty replicas, and given correctness condition C2, the client will contact at most  $f$  faulty replicas as well) and that replica will return that write, unless it was overwritten with a write with a higher timestamp (since data items are only overwritten by writes with higher timestamps at nonfaulty replicas, by algorithm construction). Since a state transfer read outputs the highest valid timestamp it sees, the lemma is true.  $\square$

**Lemma 2** For any epoch  $e$ , there are at most  $f$  nonfaulty replicas in epoch  $e$  that skip state transfer to epoch  $e$ .

**Proof.** This is true since a nonfaulty replica only skips state transfer if the MS reports the epoch to be inactive, and this can only happen after  $2f + 1$  replicas in epoch  $e$  have claimed to have completed state transfer.  $\square$

**Lemma 3** If, for at least  $f + 1$  non-faulty replicas  $i$  in the configuration for epoch  $e + 1$ , state transfer between epochs  $e$  and  $e + 1$  yields  $timestamp(st_i) \geq t$ , then every state transfer for a non-faulty replica  $j$  in a subsequent epoch will yield  $timestamp(st_j) \geq t$ .

**Proof.** Nonfaulty replicas will only overwrite the data value they read during state transfer with another value if it has a higher timestamp. Therefore, when state transfer occurs for subsequent epochs, at least  $f + 1$  nonfaulty replicas will be contacted and at least one of those replicas will output data with timestamp  $t$  or higher (given quorum intersection and correctness condition C1).  $\square$

**Lemma 4** For a given operation,  $\pi$ , if  $epoch(\pi) = e$ , then for every operation  $\phi$  such that the response event for  $\pi$  precedes the invocation event for  $\phi$ , no phase of  $\phi$  completes in epoch  $e'$ ,  $e' < e$ .

**Proof.** Assume, for the sake of contradiction, that operation  $\pi$  precedes operation  $\phi$ , and one of the phases of  $\phi$  completes in an earlier epoch. If  $epoch(\pi) = e$ , then all nonfaulty replicas in quorum  $Q$  where the last phase of the operation completed had upgraded to epoch  $e$  at the time the reply was produced. By algorithm construction, in order for these replicas to produce a reply, they must have successfully completed state transfer from epoch  $e - 1$ . The same argument applies to the replicas in  $e - 1$ : for state transfer to occur from  $e - 1$  to  $e$ , the replicas in  $e - 1$  must have already successfully completed state transfer from epoch  $e - 2$ , and successively until we reach epoch  $e'$ . Thus we conclude that by the time operation  $\pi$  was executed,  $2f + 1$  replicas of epoch  $e'$  had upgraded, at least to epoch  $e' + 1$ .

Now consider operation  $\phi$  that is invoked subsequently. By correctness condition C2, the quorums contacted in operation  $\phi$  contain at least  $f + 1$  nonfaulty replicas, which execute one of the phases in epoch  $e'$ . Since nonfaulty replicas will not downgrade their epoch numbers and do not execute requests for a stale epoch (by algorithm construction), this contradicts the fact that  $2f + 1$  replicas of epoch  $e'$  had upgraded to the next epoch.  $\square$

**Lemma 5** All timestamp values for distinct write operations are distinct.

**Proof.** For writes by different clients this is true due to timestamp construction. For writes by the same client, by well-formedness (as defined in [60]), these operations occur sequentially. So all we must argue is that the read phase of the latest write “sees” the timestamp of the previous write, or a higher timestamp. If the write phase

of the first write and the read phase of the second write complete in the same epoch, then this is true due to quorum intersection (the write quorum for the first write and the read quorum for the first phase of the second write intersect in at least one non-faulty replica, given correctness condition C2) and because replicas store increasing timestamps. Therefore, that non-faulty replica stored a timestamp greater or equal than the one for the first write, and due to the way timestamps are chosen for write operations the timestamp chosen for the later write is greater than the timestamp stored by that replica. Now suppose that the write phase of the first write executed in epoch  $e$ , and the read phase of the second write executed in epoch  $e' \neq e$ . By Lemma 4, it must be that case that  $e' > e$ . In this case, Lemmas 1, 2, and 3 will imply that  $f + 1$  nonfaulty replicas in epoch  $e$  will execute state transfer for epoch  $e'$  and read a timestamp greater or equal than the timestamp of the first write. Since nonfaulty replicas never overwrite with a smaller timestamp, the read phase will contact at least  $f + 1$  nonfaulty replicas (given correctness condition C2), and, by quorum intersection, at least one of them will present a valid timestamp greater or equal than the timestamp of the first write, and the same argument as above applies.  $\square$

**Theorem 1** The algorithm described before is a read/write atomic object [60].

**Proof.** Well-formedness, as defined in [60], is easy to see.

For atomicity, we use Lemma 13.16 of [60]. We define a partial ordering on operations in  $\Pi$ . Namely, we say that  $\pi \prec \phi$  if either of the following applies:

1.  $timestamp(\pi) < timestamp(\phi)$
2.  $timestamp(\pi) = timestamp(\phi)$ ,  $\pi$  is a *write*, and  $\phi$  is a *read*

where  $timestamp(\pi)$  is defined as the timestamp written in the second phase of operation  $\pi$ , or unanimously read in the first phase of a read.

It is enough to verify that this satisfies the four conditions needed for Lemma 13.16 of [60].

1. For any operation  $\pi \in \Pi$ , there are only finitely many operations  $\phi$  such that  $\phi \prec \pi$ .

Suppose for the sake of contradiction that operation  $\pi$  has infinitely many  $\prec$  predecessors. Lemma 5 implies that it cannot have infinitely many predecessors that are *write* operations, so it must have infinitely many predecessors that are *read* operations. (Without loss of generality, we may assume that  $\pi$  is a *write*.) Then there must be infinitely many *read* operations with the same timestamp,  $t$ , where  $t$  is smaller than  $timestamp(\pi)$ . But the fact that  $\pi$  completes in the execution of the algorithm implies that  $timestamp(\pi)$  gets written to a quorum in an epoch  $e$ . After this happens, any *read* operation that is subsequently invoked either completes its read phase in epoch  $e$  or in an epoch  $e' > e$  (by Lemma 4). If it completes the read phase in epoch  $e$ , it will read  $timestamp(\pi)$  or a higher timestamp, due to quorum intersection and the fact that replicas store increasing timestamps. If it completes in an epoch  $e' > e$ , Lemmas 1, 2, and 3, and the fact that nonfaulty replicas store increasing timestamps imply that the read phase in epoch  $e'$  will read  $timestamp(\pi)$  or a higher timestamp. This contradicts the existence of infinitely many *read* operations with timestamp  $t$ .

2. If the response event for  $\pi$  precedes the invocation event for  $\phi$  in any sequence of actions of the system, then it cannot be the case that  $\phi \prec \pi$ .

By Lemma 4, both phases of  $\phi$  must complete in an epoch greater or equal than the epoch in which  $\pi$  completes. Suppose that the write phase of  $\pi$  completes in the same epoch as the read phase of  $\phi$ . In this case, quorum intersection will ensure that the read phase of  $\phi$  will see a value greater or equal than  $timestamp(\pi)$  which suffices to show the implication above for this case. (Note that some operations do not have a write phase, but only if it is a read where all replicas agree on the timestamp, in which case the same argument applies to the unanimous read phase of  $\pi$  instead of its write phase.)

Otherwise, the last phase of  $\pi$  executes in an epoch  $e$  earlier than the epoch of the read phase of  $\phi$ ,  $e' > e$ . In this case, Lemmas 1, 2, and 3 will imply that state transfer for epoch  $e'$  read a timestamp greater or equal than the timestamp of the first write (in at least  $f + 1$  nonfaulty replicas). Since nonfaulty replicas never overwrite with a smaller timestamp, and given correctness condition C2, the read phase will contact

at least  $f + 1$  nonfaulty replicas, at least one of which will present a valid timestamp greater or equal than the  $timestamp(\pi)$ , and the same argument as above applies.

3. If  $\pi$  is a *write* operation and  $\phi$  is any operation in  $\Pi$ , then either  $\pi \prec \phi$  or  $\phi \prec \pi$ .

By Lemma 5, all *write* operations obtain distinct timestamps. This implies that all of the *writes* are totally ordered, and also that each *read* is ordered with respect to all the *writes*.

4. The value returned by each *read* operation is the value written by the last preceding *write* operation according to  $\prec$  (or  $v_0$ , if there is no such *write*).

Let  $\pi$  be a *read* operation. The value  $v$  returned by  $\pi$  is just the value that  $\pi$  finds associated with the largest timestamp,  $t$ , among the replies in quorum  $Q$ . This value also becomes the timestamp associated with  $\pi$ . By the unforgeability of data items (i.e., data is signed thus faulty replicas cannot produce forged data), there can only be two cases:

- Value  $v$  has been written by some *write* operation  $\phi$  with timestamp  $t$ . In this case, the ordering definition ensures that  $\phi$  is the last *write* preceding  $\pi$  in the  $\prec$  order, as needed.
- $v = v_0$  and  $t = 0$ . In this case, the ordering definition ensures that there are no *writes* preceding  $\pi$  in the  $\prec$  order, as needed. □

# Appendix D

## Multi-Party Secure Coin Tossing Scheme

This appendix presents the multi-party secure coin tossing scheme we implemented. Section D.1 describes the algorithm and Section D.2 analyzes its security.

### D.1 Algorithm Description

The algorithm to choose a random number works as follows. After the STOP operation has been executed, each MS replica invokes the CHOICE operation on the MS. The operation has as an argument a *label* that represents a random value,  $r_i$ ; the label doesn't reveal the value and it allows the value to be validated when it is produced later. E.g., the label might be a SHA-1 hash of the value.

These operation calls are ordered by the MS (via BFT). As soon as  $2f_{MS} + 1$  CHOICE operations, from different replicas, have completed, the MS is ready to move to the second phase of the protocol. (BFT authenticates the caller of a request and therefore the replicas in the MS know when this condition holds).

In the second part of the protocol, all replicas whose values were accepted in the first phase invoke the EXPOSE-VALUE operation, providing the random value,  $r_i$ , they had chosen initially. As soon as  $f_{MS} + 1$  of these operations complete, from different replicas, and where the value matches the label provided by that replica in the first

step of the protocol, the protocol is complete. At this point at least  $2f_{MS} + 1$  replicas know the same  $f_{MS} + 1$  values and can produce the nonce for the next epoch. This nonce is  $h(r_1, \dots, r_{f_{MS}+1})$ , where  $h$  is the SHA-1 hash function, and the  $r_i$  are the first  $f_{MS} + 1$  values that were received, in id order. The properties of the BFT state machine replication algorithm ensure that all replicas see the same outcome for these operations.

The soundness of our scheme relies on the fact that malicious nodes have to commit to a random value before good nodes disclose their random value. However, because we have an asynchronous network model and have to decide after receiving  $f_{MS} + 1$  random values, the adversary can choose among a small number of different random values the one is most convenient for it. For instance, an attacker that controls  $f_{MS} > 1$  node can propose  $f_{MS}$  hashes, wait until  $f_{MS}$  good nodes disclose their values, and then decide among the  $f_{MS}$  hashes it controls which one to propose first.

## D.2 Soundness Analysis

Here, we discuss the probability that the adversary gets control of the MS. We make several assumptions. First, the adversary controls at most  $f_{MS}$  nodes in the MS at the beginning of the epoch. Second, the adversary cannot break the security of the commitments, either to change his own commitments or to read the commitments of the honest party. Third, we assume the hash function SHA-1 cannot be distinguished from a random oracle. The first assumption is essentially our inductive assumption: we will show that this remains true at the end of the epoch. The second assumption should be true if the cryptographic primitives are good ones. The last assumption, that a hash function is a random oracle, is one that is controversial in some ways in the cryptographic community but it is well recognized that this assumption is reasonable in practice [9].

If these assumptions hold, the adversary may at best choose which  $f_{MS} + 1$  values are used as the inputs to the hash function. This means the adversary can choose



one of many random outputs from the hash function. The adversary gets one output from every subset of size  $f_{MS} + 1$  of the random values from the set of  $2f_{MS} + 1$  values. In other words, the adversary gets  $\binom{2f_{MS}+1}{f_{MS}+1}$  choices.

We can see that

$$\begin{aligned}
\binom{2n+1}{n+1} &= \binom{2n+1}{n} \\
&= \binom{2n-1}{n-2} + 2\binom{2n-1}{n-1} + \binom{2n-1}{n} \\
&= \frac{n-1}{n} \binom{2n-1}{n-2} + 3\binom{2n-1}{n-1} \\
&= 4\binom{2n-1}{n} - \frac{1}{n} \binom{2(n-1)+1}{(n-1)+1} \\
&\leq 4\binom{2(n+1)-1}{(n+1)-1},
\end{aligned}$$

and therefore,  $\binom{2f_{MS}+1}{f_{MS}+1} = O(4^{f_{MS}})$ . Thus, the adversary in effect gets to control about  $2f_{MS}$  bits of entropy of the final choice without breaking any of the cryptographic primitives. Nonetheless, this amount of control of the nonce is not enough to let the adversary have a good chance of corrupting the MS. The total entropy of choosing the MS for the next epoch from all  $n$  nodes is at least  $(3f_{MS} + 1) \log_2 n - (3f + 1) \log_2(3f + 1)$ , so after the subtracting  $2f_{MS}$  from this, there is still at least  $(3f_{MS} + 1) \log_2(n/(2^{2/3})) - (3f + 1) \log_2(3f + 1)$  entropy remaining. In other words, at worst, it is still like making a random choice of  $3f + 1$  elements out of  $n/(2^{2/3})$  possible choices.

It is easy to check that if the adversary has a total of  $k$  corruptions in the whole system, then the probability that when we choose  $3f_{MS} + 1$  machines out of  $n/(2^{2/3})$  choices including all  $k$  corruptions, the probability the adversary succeeds is upper bounded by  $\binom{3f_{MS}+1}{f_{MS}+1} (2^{2/3}k/n)^{f+1}$ . If we fix a proportion  $\alpha$  so that we assume  $k \leq \alpha n$  then this is upper bounded by  $\binom{3f_{MS}+1}{f_{MS}+1} (2^{2/3}\alpha)^{f+1}$ . Using a similar technique as before, we can show that  $\binom{3f_{MS}+1}{f_{MS}+1}$  grows slower than  $8^{f_{MS}}$  so if we pick  $\alpha < \frac{1}{8(2^{2/3})} \approx .0787$  then this probability approaches 0 exponentially as  $f$  grows. This shows that by choosing the system parameters appropriately, we can keep any adversary from corrupting the MS.



# Bibliography

- [1] TCPA – Trusted computing platform alliance. <http://www.trustedcomputing.org/>.
- [2] CERT/CC Statistics 1988-2004. [http://www.cert.org/stats/cert\\_stats.html](http://www.cert.org/stats/cert_stats.html), 2004.
- [3] Google press center: Technical highlights. [http://www.google.com/intl/cs\\_ALL/press/highlights.html](http://www.google.com/intl/cs_ALL/press/highlights.html), 2004.
- [4] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.
- [5] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, San Francisco, CA, October 1976.
- [6] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael Reiter, and Rebecca Wright. Dynamic Byzantine Quorum Systems. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN)*, pages 283–292, New York, New York, June 2000.

- [7] David Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, October 2001.
- [8] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.
- [9] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st Conference on Computer and Communications Security*, pages 62–73, 1993.
- [10] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1987.
- [11] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Replication strategies for highly available peer-to-peer storage systems. Technical Report CS2002-0726, UCSD, November 2002.
- [12] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, February 2003.
- [13] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey Voelker. Total recall: System support for automated availability management. In *Proceedings of the First ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, United States, March 2004.
- [14] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology - CRYPTO'99*, pages 216–233, 1999.
- [15] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop

- PCs. In *Proceedings of the international conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 34–43, 2000.
- [16] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th annual ACM symposium on Principles of distributed computing (PODC)*, Portland, Oregon, United States, July 2000.
- [17] Miguel Castro. *Practical Byzantine Fault-Tolerance*. PhD thesis, Massachusetts Institute of Technology, November 2000.
- [18] Miguel Castro, Peter Druschell, Ayalvadi Ganesh, Antony Rowstron, and Dan Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, Massachusetts, December 2002.
- [19] Miguel Castro and Barbara Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.
- [20] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, Louisiana, United States, February 1999.
- [21] Miguel Castro and Barbara Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.
- [22] Benjie Chen and Robert Morris. Certifying program execution with secure processors. In *Proceedings of The Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, Hawaii, United States, May 2003.

- [23] Kathryn Chen. Authentication in a reconfigurable byzantine fault tolerant system. Master's thesis, Massachusetts Institute of Technology, July 2004.
- [24] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.
- [25] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, October 2001.
- [26] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the First ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, March 2004.
- [27] Roberto de Prisco, Alan Fekete, Nancy Lynch, and Alexander Shvartsman. A dynamic primary configuration group communication service. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 64–78, September 1999.
- [28] Danny Dolev, Idit Keidar, and Esti Yeger Lotem. Dynamic voting for consistent primary components. In *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing*, pages 63–71, August 1997.
- [29] John Douceur. The sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, Massachusetts, United States, March 2002.
- [30] Burkhard Englert and Alexander Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proc. International Conference on Distributed Computer Systems (ICDCS'2000)*, pages 454–463, 2000.
- [31] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of*

- the ACM*, 19(11):624–633, November 1976. Also published as IBM RJ1487, December, 1974.
- [32] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.
- [33] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP'03)*, October 2003.
- [34] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Conference*, December 2002.
- [35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.
- [36] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, December 1979. ACM SIGOPS.
- [37] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *Sigact News*, 33(2), 2002.
- [38] Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 259–268, San Francisco, California, USA, June 2003.
- [39] Jim N. Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

- [40] Steven Gribble, Eric Brewer, Joseph Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.
- [41] Rachid Guerraoui and André Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, 2001.
- [42] Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.
- [43] Maurice P. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.
- [44] Maurice P. Herlihy and Jeannette M. Wing. Axioms for Concurrent Objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 13–26, Munich, Germany, January 1987.
- [45] Amir Herzberg, Markus Jakobsson, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive public key and signature systems. In *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*, pages 100–110, 1997.
- [46] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [47] Sushil Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *Transactions on Database Systems*, 15(2):230–280, 1990.
- [48] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching pro-



- protocols for relieving hot spots on the World Wide Web. In *Proc. 29th Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, May 1997.
- [49] David Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *ACM Symposium on Parallelism in Algorithms and Architectures*, Barcelona, Spain, June 2004.
- [50] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proceedings of the Hawaii International Conference on System Sciences*, January 1998.
- [51] Lei Kong, Arun Subbiah, Mustaque Ahamad, and Douglas M. Blough. A reconfigurable byzantine quorum approach for the Agile store. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS)*, October 2003.
- [52] John Kubiawicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, 2000.
- [53] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
- [54] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 1986.
- [55] Leslie Lamport. The Part-Time Parliament. Report Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

- [56] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.
- [57] Edward Lee and Chandramohan Thekkath. Petal: Distributed virtual disks. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 84–92, 1996.
- [58] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 226–238, Pacific Grove, California, 1991.
- [59] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. LH\* — a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.
- [60] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
- [61] Nancy Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in content addressable networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, Massachusetts, United States, March 2002.
- [62] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, pages 173–190, October 2002.
- [63] Nancy Lynch and Alexander Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Twenty Seventh Annual International Symposium on Fault Tolerant Computing*, pages 272–281, June 1997.

- [64] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
- [65] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.
- [66] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS)*, pages 51–58, October 1998.
- [67] Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic byzantine storage. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN)*, June 2004.
- [68] David Mazières. A toolkit for user-level file systems. In *Proc. Usenix Technical Conference*, pages 261–274, June 2001.
- [69] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology*. 1987.
- [70] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, Massachusetts, December 2002.
- [71] Brian Oki and Barbara Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.
- [72] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, Washington, United States, March 2003.

- [73] John Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the Usenix Summer 1990 Technical Conference*, pages 247–256, Anaheim, California, United States, June 1990.
- [74] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.
- [75] Larry Peterson, David Culler, Thomas E. Anderson, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, New Jersey, USA, October 2002.
- [76] Evelyn Pierce and Lorenzo Alvisi. A recipe for atomic semantics for byzantine quorum systems. Technical report, Department of Computer Sciences, The University of Texas at Austin, June 2002.
- [77] Michael O. Rabin. Randomized byzantine generals. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.
- [78] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, August 2001.
- [79] Michael Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, pages 99–110, 1995.
- [80] Michael Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.
- [81] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, California, March 2003.

- [82] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.
- [83] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
- [84] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, October 2001.
- [85] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.
- [86] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, October 2001.
- [87] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 1–15, Kiawah Island, South Carolina, United States, December 1999.
- [88] Stefan Saroiu, P. Krishna Gummadi, and Steven Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking 2002 (MMCN'02)*, January 2002.
- [89] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

- [90] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, August 2001.
- [91] Jeremy Stribling. Planetlab - all pairs pings. [http://www.pdos.lcs.mit.edu/~strib/pl\\_app/](http://www.pdos.lcs.mit.edu/~strib/pl_app/).
- [92] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.
- [93] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, Massachusetts, United States, March 2002.
- [94] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. APSS: Proactive secret sharing in asynchronous systems. Submitted for publication.