Mechanisms to Tolerate Misbehavior in Replicated Systems

Byung-Gon Chun

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-103
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-103.html

August 17, 2007

Mechanisms to Tolerate Misbehavior in Replicated Systems

by

Byung-Gon Chun

B.S. (Seoul National University, Seoul, Korea) 1994
M.S. (Seoul National University, Seoul, Korea) 1996
M.S. (Stanford University, Stanford) 2002

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor John Kubiatowicz, Chair
Professor Scott Shenker
Professor John Chuang

Fall 2007

The dissertation of Byung-Gon Chun is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2007

Mechanisms to Tolerate Misbehavior in Replicated Systems

Abstract

Mechanisms to Tolerate Misbehavior in Replicated Systems

by

Byung-Gon Chun
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Kubiatowicz, Chair

Distributed systems face challenges for operating correctly despite misbehavior of their components. Redundancy through replication is a widely-used technique to combat against misbehavior. However, this technique has a fundamental limitation in terms of the number of arbitrary faults it can tolerate. This limitation becomes a more serious problem when a system operates for long periods of time. Furthermore, components may deviate from specification because of rational behavior when they operate in different administrative domains.

In this thesis, we explore mechanisms to tolerate misbehavior of components in distributed systems, focusing on replicated systems. First, we show that equivocation – the act of telling different lies to different nodes – is a fundamental weapon that adversaries can use to violate the safety of systems. To prevent equivocation, we propose Attested Append-Only Memory (A2M), a trusted system facility that is small, easy to implement, and easy to verify. A2M provides the programming abstraction of a trusted log, which leads to protocol designs immune to equivocation. Using A2M, we improve upon the state of the art in Byzantine-fault tolerant replicated state machines, producing A2M-enabled protocols (variants of Castro and Liskov's PBFT) that remain correct (linearizable) and keep making progress (live) even when half the replicas are faulty, improving the previous upper bound. We also applied A2M to achieve linearizability in a single-server shared storage in spite of faults. Our evaluation shows that this fault tolerance improvement is achieved with minor performance overhead.

Second, we address the problem of long-term fault tolerance. Typical Byzantine models require that the number of faulty nodes do not exceed a hard upper bound. Unfortunately, in long-running systems, uninterrupted good health is tough to guarantee due to rare, short-term overwhelming faults such as malicious attacks, leading to loss of all correctness properties. To combat this problem, we propose a tiered Byzantine fault model that has two fault bounds depending on the type of operations. We introduce a desirable property called Healthy-Write-Implies-Correct-Read (HWICR) which stipulates that the system will return correct data as long as it is written during a good period of system health. We then present TimeMachine (TM), a preserved name service, that uses a two-phase approach to provide HWICR under the tiered fault model. The approach alternates between service phase and proactive recovery phase, and important state changes happen only during proactive recovery. Our prototype demonstrates that TM meets the goal of the long-term naming service with reasonable performance.

Finally, we tackle a problem of replication among rational nodes in multiple administrative domains. We take a game-theoretic approach to quantify the effects of rationality on the social cost of replicated systems. We show that replication performed by selfish agents can be very inefficient, but with a proper incentive mechanism such as payment the system can be guided to socially optimal replication.

Professor John Kubiatowicz
Dissertation Committee Chair

FOR MY FAMILY

Contents

List of Figures

Acknowledgments

I thank the Lord for blessing me with this great opportunity to study Computer Science at this intellectually-inspiring place, Berkeley, guiding me to work on great projects with great mentors, and allowing me to grow up spiritually during my study.

I am extremely lucky to have my advisors, Professor John Kubiatowicz and Professor Scott Shenker, during my Ph.D. study. In the beginning of my study, John Kubiatowicz invited me to the world of large-scale distributed systems, in particular his OceanStore project. He has advised me to formulate high-level ideas and to design detailed system algorithms and implementations. Scott Shenker inspired me to work on many interesting projects including fault tolerance and core networking technologies. He has guided me to work on important problems and to think about the impact of my work. I am also very lucky to have Dr. Petros Maniatis at Intel Research, Berkeley as my mentor during my Ph.D. study. Both A2M and TimeMachine are the results of our collaboration. He has encouraged me to pursue my work, and has given me numerous valuable feedback on the work. Professor John Chuang kindly became my dissertation committee member. My work on systems with rational nodes was greatly influenced by his advice and work. I am grateful for his commitment.

I thank my mother with my heart. Without her love, support and sacrifice, it would be impossible for me to pursue my study for many years and I'd not be who I am. I thank my father in heaven for his love. I always remember his smile. I thank my wife Chan Jean Lee, my life partner, for her constant encouragement and support throughout my academic years. Our common intellectual interests helped my research. My lovely two kids, Soomin Christin Chun and Joseph Sukmin Chun, have brought so much joy to me and my family. They love computers like I do! I thank my brother and his wife for praying for me and supporting me as a family. I especially thank my parent-in-law to help us raise our kids while they visited Berkeley as visiting scholars. I thank my grandfather and grandmother in heaven, who helped my family when my father was sick, for their endless love, I always miss them. Lastly, my mother's relatives have been a big support to me since I was a child.

My office mates, Hakim Weatherspoon, Sean Rhea, Patrick Eaton, Victor Wen, and Matthew Caesar have been great friends. Our collaboration and discussion helped me enrich my experience in designing and building distributed systems and networks. I thank the networking group at Berkeley led by Professors Scott Shenker and Ion Stoica. I also thank RADLab for providing open research space. My first year at Berkeley was great because of my office mates, Rodrigo Fonseca, Ana Ramirez Chang, and Evan Chang. Rodrigo Fonseca and I share many common research interests and he has given me valuable feedback on my research throughout my academic years. I also thank Mary Barnes for helping me focus on my study during my first year at Berkeley.

I want to thank other collaborators of this thesis. The Selfish Replication chapter is the result of collaboration with Hoeteck Wee, Kamalika Chandhuri, Marco Barreno, and Christos Papadimitriou. The analysis of maintaining replicas for durability is the result of collaboration with Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, Frans Kaashoek, and Robert Morris.

I thank my friend Sukwon Cha and Moon Jung Kim for giving me good advice and helping me settle when I started my study in the states. I am indebted to Byunghoon Kang. His family helped our family settle during our transition to Berkeley. Fellow Korean graduate EECS students, Jaein Jung, Sukun Kim, Daekyeong Moon, and Hyuck Choo, have been great support.

I thank Pastor Bae in Richmond Baptist Church for praying for our family. I thank my church cell group members for our fellowship and I thank the Lord for blessing me to be part of the current cell group.

Chapter 1

Introduction

1.1 Motivation

People rely on services provided by computer systems in everyday life. Many systems are networked through the Internet, often across multiple administrative domains. Mission critical systems such as banking, medical applications, emergency response, airline control, military applications, and space mission applications are not uncommon. For critical computer systems, reliability is the most important goal.

However, achieving reliability by tolerating faults becomes harder due to several reasons. Complexity in software and hardware has increased tremendously for decades. It is hard to write bug-free or error-free software. As software complexity increases, bug rates get worse [CYC$^+$01]. Furthermore, bugs in networked systems can open door to security attacks through the Internet. In 2006 alone, the CERT Coordination Center [cer] received more than 8,000 reports of security vulnerabilities. Often malicious attackers can use viruses or worms to infect hosts with common vulnerabilities quickly. Human errors and misconfigurations can also thwart the reliability of systems. These faults can be modeled as *Byzantine faults* where faulty nodes can behave arbitrarily.

Replication is a fundamental technique for tolerating faults. The basic idea is that multiple servers coordinate to run the same program image or to keep the same copy of data, thus giving clients an illusion of interacting with a single server. Many protocols have been designed to tolerate benign faults (i.e., fail-stop faults) [Lam01, OL88], but these protocols do not defend against Byzantine faults. However, as we point out in the above, tolerating Byzantine faults becomes more and more necessary for critical systems. Byzantine fault tolerant (BFT) protocols are evaluated by fault bounds, i.e., how many faults a protocol can tolerate before it loses correctness or liveness guarantees. In this regard, they have a fundamental limitation. It is proven in theory that they can tolerate up to less than 1/3 faulty replicas [LSP82, CL02]. Improving this fault bound means that the system can tolerate more faults before something bad happens, which is important for critical systems.

Byzantine fault tolerance becomes more challenging when a system operates for long periods of time. For example, digital preservation systems must preserve data for decades or longer. BFT protocols are problematic for long-term services since there is no guarantee on the system state for the past, the current, and the future when their fault bound is violated. However, for long-running applications, it is highly likely that they do violate the fault bound, so it is important to reduce time vulnerable to fault bound violation.

Furthermore, systems that span multiple administrative domains may suffer from rational behavior of participants, which is another kind of misbehavior. Participants in the system may not follow protocol specification to maximize their local utilities and to free ride the system. In this setting, there is no clear bound on how many participants behave rationally. Traditional BFT protocols are not suitable in this environment because of no limit on the population of rational nodes, so we need to construct protocols to tolerate rational behavior.

1.2 Problem Definition and Challenges

In the distributed systems literature, it has long been a goal to offer clients the illusion of interacting with a single, reliable, fail-stop server, despite the occurrence of Byzantine server faults. While the initial results along these lines were largely theoretical, in recent years there has been an increasing interest in producing practical Byzantine-fault tolerant systems, as exemplified by PBFT [CL02], Q/U [AEMGG+05], Ivy [MMGC02], Plutus [KRS+03], SUNDR [LKMS04], HQ [CML+06], and Zyzzyva [KAD+07].

The fault-tolerance properties of such systems can be divided into *safety* guarantees, properties that must be true at all times, and *liveness* guarantees, properties that must become true within finite time from all execution states of the system. For replicated state machines (e.g., PBFT, Q/U, HQ, and Zyzzyva) the target safety guarantee is *linearizability* [HW90]: completed client requests appear to have been processed in a single, totally ordered, serial schedule that is consistent with the order in which clients submitted their requests and received their responses. The corresponding liveness guarantee is that a correct client's request is eventually processed. It is well established that if servers have no trusted components, then no replicated system can provide these safety and liveness guarantees when more than a third of its replicas are faulty.

In deterministic systems that aim to guarantee linearizability, lying is bad enough, but lying in different ways to different nodes is much worse. The "prototype" problem behind Byzantine-fault tolerant agreement, the "Byzantine generals problem," has been demonstrated unsolvable among 3 parties when one is faulty [LSP82]. The proof involves a situation in which a faulty node equivocates, namely gives different information to each of the other nodes. Even with a single server, equivocation can wreak havoc: a faulty server can order sequential requests in different ways when responding to different clients, potentially changing the presumed state of the system substantially. For the case of two conflicting writes a and b, this could result in one client seeing a as the dominating write whereas the other client sees b instead. Thus, equivocation is a fundamental problem that limits Byzantine fault tolerance.

BFT models require that the number of faulty nodes do not exceed a hard upper bound, such as a third of the entire population of nodes. It has been argued that such a bound is maintainable in a reliable replicated service if participating nodes are well enough managed, secured, and maintained that they can mostly avoid network-triggered exploits of unpatched bugs and the physical manipulation of malicious humans. In such a setting, it appears reasonable to provide the usual correctness property: as long as the system is *healthy* – that is, no more than f out of $3f+1$ nodes are faulty at any point in time – the system will offer its correctness guarantees [ZSR02, CL02].

Unfortunately, in long-running systems such as digital preservation, uninterrupted good health is tough to guarantee. First, malicious attacks such as virus and worm infections are increasingly hard to stop, even in well managed enterprise settings; the fact that most nodes in a replicated system will be running one or perhaps two distinct implementations and operating systems, prone to the same exploits, does not help the situation either. Second, after decades of continual use, human errors, organizational slip-ups, and other unlikely events are bound to crop up [BSR$^+$06], causing bound violations to occur. Even if one such slip into an *unhealthy period* occurs, the correctness of typical BFT systems can no longer be guaranteed, not just for the duration of the violation, but also forever into the future. For example, in a system such as Castro and Liskov's Practical BFT (PBFT) [CL02], once the fault bound is violated, faulty nodes can cause non-faulty nodes to execute distinct, divergent sequences of operations on their local states, from which they cannot recover without human intervention [LM07].

To overcome failures in a single domain, systems can replicate content across multiple administrative domains. Examples are PlanetLab [BBC$^+$04], the Global Information Grid [Age], and GRID. Most such systems assume that servers cooperate with one another by following protocols optimized for overall system performance, regardless of the costs incurred by each server. In reality, servers may behave rationally — seeking to maximize their own benefit. For example, parties in different administrative domains utilize their local resources (servers) to better support clients in their own domains. They have obvious incentives to replicate objects that maximize the benefit in their domains, possibly at the expense of globally optimum behavior. They also have incentives to gain their service without spending their resources, which is *free riding*. Therefore, it is necessary to address whether these replication scenarios and protocols maintain their desirable global properties (low total social cost, for example) in the face of rational behavior.

In this thesis, we explore a set of mechanisms to improve misbehavior tolerance of replicated systems. In particular, we investigate improving the fundamental Byzantine fault bound in both short-term and long-term services and mitigating the effect of rational behavior on the costs of replicated systems.

1.3 Contribution

Addressing concerns of the previous sections, this thesis provides the following contributions.

System Support for Improving Byzantine Fault Tolerance: We argue that a *trusted log* abstraction, which we call Attested Append-Only Memory or A2M for short, can improve the fault tolerance of systems in the face of Byzantine faults. A2M is a small-footprint trusted primitive that has a simple interface, is broadly applicable, and can be implemented easily and cost effectively. The power of A2M lies in its ability to eliminate *equivocation*, telling different stories to different entities, from the possible failure modes of untrusted components; that is, a faulty replica in a replicated system cannot undetectably answer the same question with different answers to different clients or other replicas.

Using A2M, we construct A2M protocols that achieve stronger guarantees than previous protocols provide. In particular, A2M-Storage achieves linearizability when a file system is shared by multiple clients on a untrusted server. We present two variants of Practical Byzantine Fault Tolerance (PBFT) [CL02] that improve the fundamental Byzantine fault bound. Similar to PBFT, A2M-PBFT-E guarantees safety and liveness with up to $\lfloor \frac{N-1}{3} \rfloor$ faulty replicas out of N total; however, whereas PBFT offers no guarantees whatsoever when this upper bound of faulty replicas is crossed, A2M-PBFT-E can still guarantee safety without liveness when faulty replicas are more than $\lfloor \frac{N-1}{3} \rfloor$ but no more than $2\lfloor \frac{N-1}{3} \rfloor$. A2M-PBFT-EA is an extension of PBFT that can guarantee both safety and liveness with up to $\lfloor \frac{N-1}{2} \rfloor$ replica faults by protecting PBFT agreement and execution steps. We also show A2M is applicable to quorum-based state machine replication.

Long-term Fault Tolerance: We study Byzantine fault tolerance for long-term services such as digital preservation. We pinpoint challenges and problems in traditional BFT protocols and propose a new service property for long-term services. We introduce the Healthy-Write-Implies-Correct-Read (HWICR) property, which states that once a value is written during a good system period it is correctly read afterwards (i.e., the system never returns an incorrect value) despite intervening bad system periods that violate traditional fault assumptions. To achieve HWICR, we propose a more realistic fault model, which we call tiered Byzantine fault model, than traditional Byzantine fault models. In the tiered fault model, we divide operations into regular operations and trusted operations. For regular operations, we assume no more than $N-1$ nodes are faulty (i.e., there is at least one non-faulty node). For trusted operations that are performed by a more trusted component (e.g., trusted hardware), we assume no more than $\lfloor \frac{N-1}{3} \rfloor$ nodes are faulty. By dividing system behavior into two regions, we focus on how the presence of a simple trusted abstraction (operating with a traditional Byzantine fault threshold) can be used to relax the fault bounds of the overall system.

To have a concrete context, we apply our model to a long-term digital preservation service. Though durability and availability have been addressed comprehensively by systems such as OceanStore [KBC$^+$00] and Glacier [HMD05], authenticity has received less satisfying solutions: the typical approach is to rely on *self-verifying data,* for which the name of a data item is an *authenticator* for that data item, which can be used to verify the item itself (e.g., a cryptographic hash). Users who can remember such a name (a long string of otherwise meaningless digits) can ascertain long-term authenticity of the corresponding content fetched from a preservation service. This solution does not, however, deal with usage models in which a user decades down the road wishes to authenticate the contents of a preserved document or a collection of documents (e.g., “State Budget Fiscal Year 2003”, “UCB EECS Snapshot 2002-02-07”). When lookup of preserved content is by a human-readable name, existing systems provide no solution to preserving the mapping between a name and an authenticator for a data item, assuming instead that this is done by some trusted third party.

We address this naming service problem by proposing *TimeMachine* (TM), which achieves HWICR under the tiered Byzantine fault model. TM uses a two-phase approach where regular service and trusted proactive recovery phases alternate. During the service phase TM serves clients’ read requests and temporarily buffers write requests. Only during the proactive recovery phase TM makes important state changes, incorporating buffered write requests to its main data store.

Rational Behavior Tolerance in Replication: We address the problem of replication in networks of selfish servers running in multiple administrative domains through theoretical analysis and simulations. We take a game-theoretic approach to analyzing this problem. We model selfish replication as a non-cooperative game. In the *basic model*, the servers have two possible actions for each object. If a replica of a requested object is located at a nearby node, the server may be better off accessing the remote replica. On the other hand, if all replicas are located too far away, the server is better off replicating the object itself. Decisions about replicating the replicas locally are arrived at locally, taking into account only local costs. We also define a more elaborate *payment model*, in which each server bids for having an object replicated at another site. Each site now has the option of replicating an object and collecting the related bids. Once all servers have chosen a strategy, each game specifies a *configuration*, that is, the set of servers that replicate the object, and the corresponding costs for all servers.

The lack of coordination inherent in selfish decision-making may incur costs well beyond what would be globally optimum. This loss of efficiency is quantified by the *price of anarchy* [KP99]. The price of anarchy is the ratio of the social (total) cost of the worst possible Nash equilibrium to the cost of the social optimum. The price of anarchy bounds the worst possible behavior of a selfish system, when left completely on its own. However, in reality there are ways whereby the system can be guided, through “seeding” or incentives, to a pre-selected Nash equilibrium. This “optimistic” version of the price of anarchy [ADTW03] is captured by the smallest ratio between a Nash equilibrium and the social optimum.

We show that pure strategy Nash equilibria exist in the basic game. In addition, we prove that Nash equilibria are not efficient by computing the (optimistic) price of anarchy under different network topologies and placement costs. Finally, we show that by adopting payments servers are incentivized to replicate data, thus leading to Nash equilibria that have socially optimal configurations. Thus, payment can be a way to combat rational behavior problems in replicated systems.

1.4 Organization

The rest of this thesis is organized as follows. In Chapter 2, we present system support for Byzantine fault tolerance, which we call Attested Append-Only Memory (A2M), to solve equivocation problems. With A2M, we construct A2M-Storage that achieves linearizability in a single server and two variants of PBFT that improve the fundamental Byzantine fault bound. We demonstrate that this improvement can be achieved with minor performance overhead. Chapter 3 presents challenges in the fault tolerance of long-term services. We propose a new service property called Healthy-Write-Implies-Correct-Read (HWICR) and a tiered Byzantine fault model for long-term services. We design TimeMachine (TM) that achieves HWICR in the tiered fault model. We extend our discussion to systems running in multiple administrative domains in Chapter 4. Using game theory we analyze replication efficiency when participating nodes behave rationally with or without an incentive scheme. In Chapter 5 we discuss related work. In Chapter 6 we conclude and discuss potential future research directions.

Chapter 2

A2M: System Support for Fault Tolerance

2.1 Overview

In the distributed systems literature, it has long been a goal to offer clients the illusion of interacting with a single, reliable, fail-stop server, despite the occurrence of Byzantine server faults. While the initial results along these lines were largely theoretical, in recent years there has been an increasing interest in producing practical Byzantine-fault tolerant systems, as exemplified by PBFT [CL02], Q/U [AEMGG$^+$05], Ivy [MMGC02], Plutus [KRS$^+$03], SUNDR [LKMS04], HQ [CML$^+$06], and Zyzzyva [KAD$^+$07].

The fault-tolerance properties of such systems can be divided into *safety* guarantees, properties that must be true at all times, and *liveness* guarantees, properties that must become true within finite time from all execution states of the system. For replicated state machines (e.g., PBFT, Q/U, HQ, and Zyzzyva) the target safety guarantee is *linearizability* [HW90]: completed client requests appear to have been processed in a single, totally ordered, serial schedule that is consistent with the order in which clients submitted their requests and received their responses. The corresponding liveness guarantee is that a correct client's request is eventually processed. It is well established that if servers have no trusted components, then no replicated system can provide these safety and liveness guarantees when more than a third of its replicas are faulty.

To improve on these results, some researchers have explored relaxed correctness properties. For instance, *fork* consistency* [LM07] is a weaker safety property than linearizability, but can be achieved when less than two thirds of the replica population are faulty. Such bounds are useless for single-server systems, because the situation is binary: the choice is only between 0% "replica" faults (the server is non-faulty) and 100% "replica" faults (the server is faulty). SUNDR showed how to achieve *fork consistency* (slightly stronger than fork*, but still weaker than linearizability) in the presence of a faulty server and non-faulty clients.

In this thesis, our goal is to understand how the fault tolerance of such systems might be improved through the use of realistic trusted abstractions. Of course, placing the entire application (operating system, application software, hardware, intervening network) into the trusted computing base trivially solves the problem, but this is totally impractical. Our focus here is on small-footprint trusted abstractions that have simple interfaces, are broadly applicable, and can be implemented easily and cost effectively. We argue that a *trusted log* abstraction, which we call Attested Append-Only Memory or A2M for short, is such an abstraction. The power of A2M lies in its ability to eliminate *equivocation*, telling different stories to different entities, from the possible failure modes of untrusted components; that is, a faulty replica in a replicated system cannot undetectably answer the same question with different answers to different clients.

Section 2.2 motivates our choice of trusted abstraction, through examples from both replicated and single-server systems. Section 2.3 presents our first contribution, A2M, in more detail, describing its interface, typical usage patterns, and implementation alternatives that trade-off efficiency for the size of the trusted computing base.

Next, we delve deeper into our second contribution: specific system designs for replicated state machines and shared storage that use A2M to improve their fault tolerance, in the context of agreement-based replicated state machines (Section 2.4) and other centralized and distributed protocols (Section 2.5). These include:

- A2M-PBFT-E is an A2M variant of Castro and Liskov's Practical Byzantine Fault Tolerance (PBFT) protocol. Similar to PBFT, A2M-PBFT-E guarantees safety and liveness with up to $\lfloor \frac{N-1}{3} \rfloor$ faulty replicas out of N total; however, whereas PBFT offers no guarantees whatsoever when this upper bound of faulty replicas is crossed, A2M-PBFT-E can still guarantee safety without liveness when the number of faulty replicas is more than $\lfloor \frac{N-1}{3} \rfloor$ but no more than $2\lfloor \frac{N-1}{3} \rfloor$. This is an important advantage for applications, such as high-volume banking, in which correctness (captured by safety) under heavy faults is desirable, even if it is not accompanied by availability (captured by the liveness property).

- A2M-PBFT-EA is an extension of PBFT that can guarantee both safety and liveness with up to $\lfloor \frac{N-1}{2} \rfloor$ replica faults: whereas PBFT needs a three-fold replication to tolerate a given number of faults, A2M-PBFT-EA needs only two-fold replication. The additional complexity of A2M-PBFT-EA may be justifiable in applications that require both low replication *and* high fault tolerance, as might be the case for critical applications with very high replication costs, such as dependable software for space missions.

- A2M-Storage is an A2M-enabled single-server storage service similar to SUNDR [LKMS04]. A2M-Storage leverages A2M to guarantee linearizability whereas SUNDR, without help from trusted components, can only provide fork consistency.

Section 2.6 presents an experimental evaluation of the A2M approach, using microbenchmarks on our implementation of A2M and two of our A2M-enabled protocols, A2M-PBFT-E and A2M-PBFT-EA. We also show macrobenchmarks on NFS running on top of A2M-PBFT-E and A2M-PBFT-EA, which suggest that the cost of using A2M to increase fault tolerance (or, conversely, reduced redundancy) is minimal: using an A2M module through a system call-like interface, the overhead of NFS on top of A2M-PBFT-EA is about 4% compared to that of NFS on top of traditional PBFT, or about 24% compared to NFS on top of an untrusted NFS server, for the benefit of reducing replication factor from 3 to 2.

We discuss the appropriate level for a trusted abstraction in Section 2.7, discuss future work in Section 2.8, and then summarize in Section 2.9.

2.2 Motivation

In this section, we detail the fundamental motivation behind our work, starting with our basic assumptions and target system environments, and continuing with specific illustrations of an adversary's power against existing systems, which will motivate our A2M design in Section 2.3, and A2M-related protocols in Sections 2.4 and 2.5.

2.2.1 Setup

We consider client-server systems where a service is accessed and shared by multiple clients connected over a public network. The service can be implemented as a single server (e.g., a file server) or multiple servers (e.g., replicated state machines). Clients request *authenticated* operations from the service, the service executes those operations, which may change the service state, and returns responses to the requesting clients.

2.2.2 Assumptions

We use standard assumptions about the network model and about cryptography. In the network, packet drops, reorderings, and duplications can occur but retransmissions of a message eventually deliver it. However, though finite upper bounds exist for message delivery and operation execution times, those bounds are not known to protocol entities. A faulty node cannot violate intractability assumptions about standard cryptography. Therefore, the adversary cannot produce pre-images or collisions for cryptographic hash functions[1] or forge previously unseen signatures for private signing keys he does not possess.

2.2.3 Fault Models

In this thesis, we consider fault models that depend on the cause of the node's misbehavior. In particular, we distinguish between two cases: (i) the node's owner is well-intentioned but unaware the node's software has been compromised by a third-party (*faulty*

[1]A one-way – or pre-image resistant – hash function h is one for which there is no polynomial-time algorithm that, given α, can find a previously unknown β such that $\alpha = h(\beta)$. A collision-resistant hash function h is one for which there is no polynomial-time algorithm that can find two values α and β for which $h(\alpha) = h(\beta)$.

application model), and (ii) the node's Byzantine behavior is because of a malicious owner instructing it to do so (*faulty operator model*). The nature of the trusted computing base is quite different in the two cases. In the first model, the trusted computing base is set up by the service owner; for instance, a bank owns all nodes and ensures, through physical security and other means, that only its nodes can provide the service. Our concern here is to combat software attacks such as worms and viruses against those centrally administered nodes. In the second model, we do not trust owners but trust a third party (e.g., a special service provider or a trusted hardware manufacturer) to set up the trusted computing base; for instance, a malicious storage server can manipulate all aspects of its node except what lies within the trusted device, which is the purview of the device provider.

In the traditional Byzantine-fault model, the cause of Byzantine behavior is not of immediate consequence – that is, tolerant protocols work well regardless of whether the operator or a virus writer are doing the misbehaving. Nevertheless, the practical decision to apply or not a solution to a target environment depends exactly on whether the designer can explain why the Byzantine-fault bound will not be violated; the justification is dependent on whether that environment consists of a single administrative domain (benign operator, potential software attacks) or multiple administrative domains (potentially malicious operators, potential software attacks).

2.2.4 Notation

For conciseness, throughout the thesis we use the authentication notation of Yin et al. [YMV$^+$03], according to which we denote by $\langle X \rangle_{S,D,k}$ an authentication certificate that any node in a set D can regard as proof that k distinct nodes in S said X. For example, a traditional digital signature on X from p that is verifiable by the entire replica population R would be $\langle X \rangle_{p,R,1}$, two signatures from p and q put together would be $\langle X \rangle_{\{p,q\},R,2}$, and a MAC from p to q with a shared key would be $\langle X \rangle_{p,q,1}$. As a convention, we use p to denote the singleton set $\{p\}$, and ∞ as shorthand for the universal set of all principals. When we use this notation to describe collective certificates made up of individual signatures, as for the second example above, we usually remove any signer identification from the collective certificate format: for example, the certificate $\langle X \rangle_{\{p,q\},R,2}$ above could correspond to the individually signed messages $\langle p, X \rangle_{p,R,1}$ and $\langle q, X \rangle_{q,R,1}$.

We use $h()$ to denote a one-way collision-resistant hash function such as SHA-256, and $||$ to denote the bit-string concatenation operator.

2.2.5 Equivocation

In deterministic systems that aim to guarantee linearizability, lying is bad enough, but lying in different ways to different people is even worse. The "prototype" problem behind Byzantine-fault tolerant agreement, the "Byzantine generals problem," has been demonstrated unsolvable in a population of three parties when one is faulty [LSP82]. The proof involves situation in which a faulty node equivocates; namely gives different information to each of the other nodes. Even with a single server, equivocation can wreak havoc: a faulty server can order sequential requests in different ways when responding to different clients, potentially changing the presumed state of the system substantially. For the case of two conflicting writes a and b, this could result in one client seeing a as the dominating write whereas the other client sees b instead. Thus, equivocation is a fundamental problem that limits Byzantine fault tolerance.

In what follows, we present two detailed examples of equivocation attacks against single-server and replicated systems, to motivate our focus on eliminating equivocation through trusted system abstractions.

Servers Equivocating to Clients

We consider a log-structured storage server shared by multiple clients as an illustrative example. For example, in a straw-man design for SUNDR [LKMS04], to request an operation, a client first acquires a lock at the server and downloads the entire operation log, a time-ordered collection of signed client operations. The client checks whether the log is correct by verifying the signatures and by checking that the log contains all of its own operations in order; it then creates what must be the server's current state by starting with an initial state and then applying the logged operations in order, as a correct server would have in a linearized system. It executes its operation based on the constructed state, thus finding out the result of this operation. It then appends its signed operation to the end of

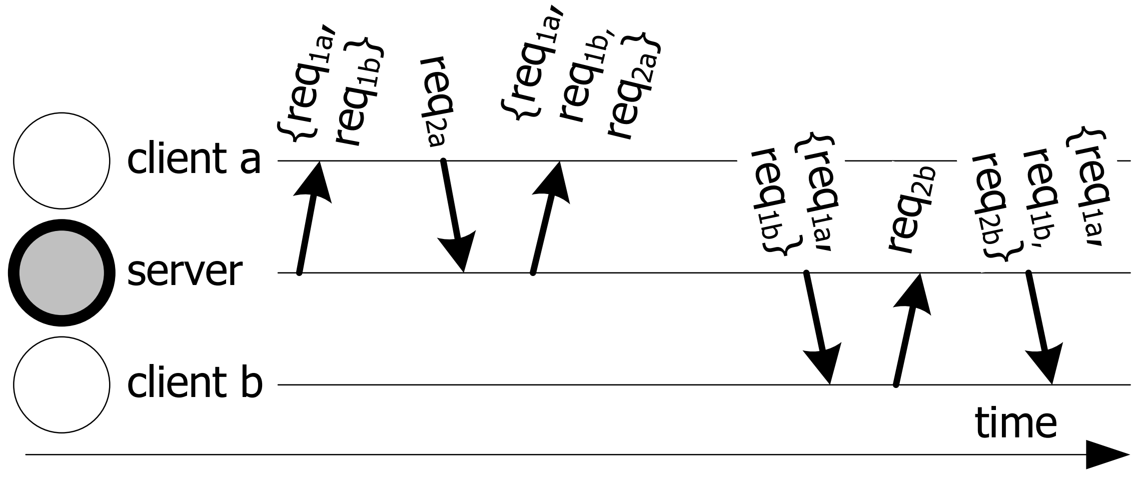

Figure 2.1: A forking attack example of two clients and one malicious server. The server convinces clients *a* and *b* of different system states.

the log, sends the updated log back to the server, and releases the lock.

A faulty server can mount a forking attack [LKMS04] by concealing operations, which causes the system's state to diverge into multiple possibilities for different clients. Suppose two clients access a server as shown in Figure 2.1. Client *a* performs req_{1a}, client *b* performs req_{1b}, and client *a* performs req_{2a}. The latest state of the server becomes $\{req_{1a}, req_{1b}, req_{2a}\}$ as far as client *a* is concerned. Now, client *b* retrieves the log of the server to perform a new operation req_{2b}. The faulty server drops req_{2a} off the tail of the log, only returning $\{req_{1a}, req_{1b}\}$. Client *b* executes its operation and has the log state $\{req_{1a}, req_{1b}, req_{2b}\}$. The system state is now forked with regards to these two clients. The cause of the problem is the ability of the faulty server to misrepresent its operation log to the two clients, equivocating on what its state is according to who is asking.

Systems vulnerable to this kind of equivocation attacks are shared file systems such as Plutus [KRS$^+$03], SUNDR [LKMS04], and Ivy [MMGC02], quorum-based replicated state machines such as Q/U [AEMGG$^+$05], and timestamping systems such as Timeweave [MB02]. SUNDR and Timeweave alleviate the effects of equivocation, offering fork consistency, a weaker property than linearizability. For example, SUNDR maintains state about the server's timeline at individual clients; once forked, all clients within the

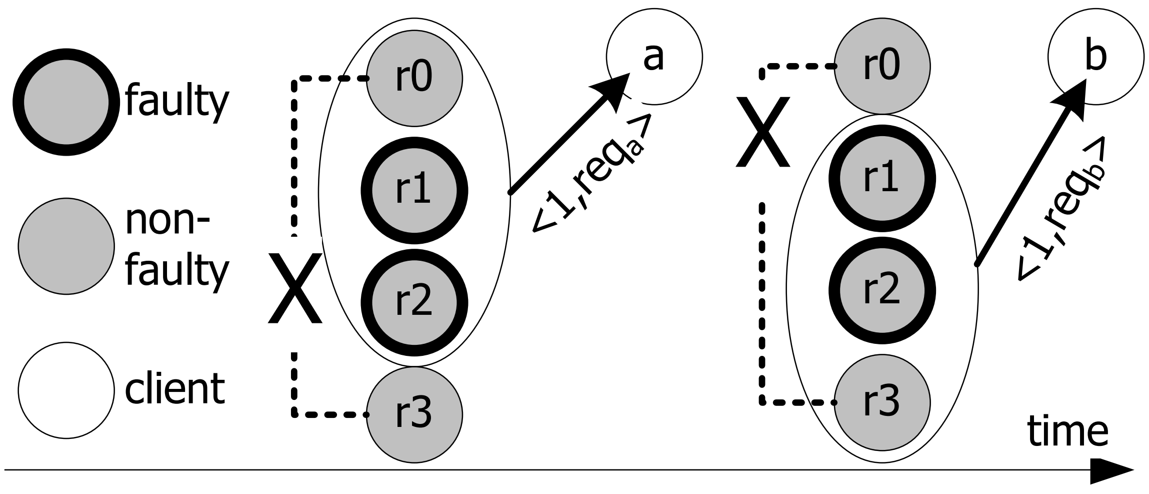

Figure 2.2: An example that shows the violation of linearizability in PBFT when two replicas are faulty out of four replicas. Faulty servers r_1 and r_2 convince non-faulty servers r_0 and r_3 to commit different requests.

same fork enjoy a linearized view of the system, but do not see state changes in another fork. Unfortunately, even then, unless two clients on different forks compare their notes, they cannot know that the server maintains multiple versions of its state and history.

Servers Equivocating to Servers

To demonstrate equivocation problems among servers, we consider BFT replicated state machines. In particular, we choose Practical Byzantine Fault Tolerance (PBFT) [CL02] since it has had a profound impact on the systems literature. Though we give more detailed background on PBFT in Section 2.4.1, for the purposes of this illustration, a PBFT client is satisfied with a result to its request if it receives at least $\lfloor \frac{N-1}{3} \rfloor + 1$ replies from distinct replicas out of the N total replicas, all with a matching result; a PBFT replica can commit a request to its local state as long as a quorum of $2\lfloor \frac{N-1}{3} \rfloor + 1$ replicas agree on the request's ordering in history.

Given this behavior, PBFT guarantees safety (linearizability) and liveness, as long as no more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty; if more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty, PBFT does not guarantee safety (and liveness is meaningless without safety): faulty replicas can fool non-faulty replicas to commit different request histories, and different clients may accept replies corresponding to different request histories, violating linearizability.

To illustrate, consider $N = 4$; replicas r_1 and r_2 are faulty, and non-faulty replicas r_0 and r_3 cannot temporarily communicate with each other (Figure 2.2). Client a sends req_a to the system. The two faulty replicas convince r_0 to commit and execute req_a first, since the three of them form a quorum of $3 = 2\lfloor \frac{N-1}{3} \rfloor + 1$. Later client b sends req_b to the system. The two faulty replicas convince r_3 to commit and execute req_b first, since r_3 never saw req_a. Faulty servers r_1 and r_2 equivocate to non-faulty servers r_0 and r_3.

Furthermore, the ability of faulty servers to equivocate to non-faulty servers also allows the service to equivocate to clients, as in the previous section. For example, clients a and b experience via their accepted replies two different histories, in which req_a and req_b are, respectively, the single, first committed request, violating linearizability. The problem arises because of the faulty replicas equivocating to clients. The faulty replicas are allowed to tell client a, with r_0's help, that req_a is committed in their history at sequence number 1, and also to tell client b, with r_3's help, that req_b is committed in their history at the same sequence number.

Systems vulnerable to servers equivocating to servers are agreement-based Byzantine-fault tolerant state machine replication protocols such as PBFT [CL99] and BFT2F [LM07]. BFT2F supports fork* consistency by maintaining state at clients.

2.3 Attested Append-Only Memory

In the previous section, we argued that the adversary's ability to equivocate undetected – e.g., to claim to have two different histories depending on which host it is talking to – is a fundamental weapon against safety, both in single-server and replicated services. Here we describe an *attested append-only memory* (A2M), a simple attestation-based abstraction that, when trusted, can remove the ability of adversarial replicas to equivocate without detection. Using an A2M implementation within the trusted computing base, a protocol can assume that a seemingly correct host can give only a single response to every distinct protocol request – for some protocol specific definition of "distinct" request –, even when that same request is retransmitted multiple times by different clients or replicas, and even

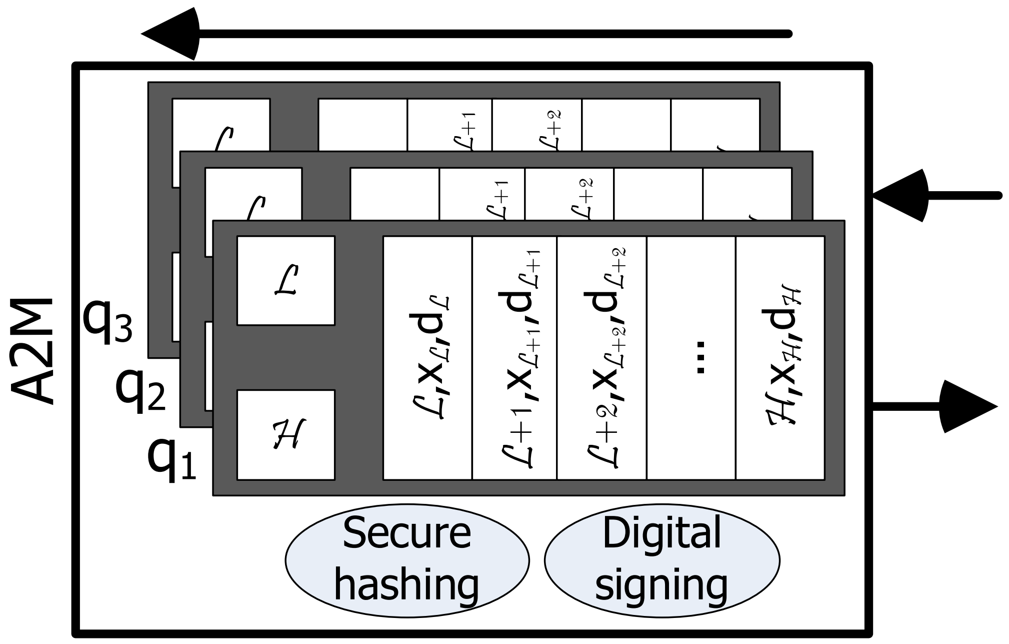
A2M
q3
q2
q1
$\mathcal{L}$
$\mathcal{H}$
$\mathcal{L}$,x$_\mathcal{L}$,d$_\mathcal{L}$
$\mathcal{L}$+1,x$_{\mathcal{L}+1}$,d$_{\mathcal{L}+1}$
$\mathcal{L}$+2,x$_{\mathcal{L}+2}$,d$_{\mathcal{L}+2}$
...
$\mathcal{H}$,x$_\mathcal{H}$,d$_\mathcal{H}$
Secure hashing
Digital signing

Figure 2.3: Structure of an *attested append-only memory* (A2M). An A2M contains a set of distinct logs (q_i) that map sequence numbers (in the range of $\mathcal{L}_i$ to $\mathcal{H}_i$) to values.

if that response is undetectably faulty.

Informally, an A2M equips a host with a set of trusted, undeniable, ordered logs (illustrated in Figure 2.3). Each such log has an identifier q (unique within the same computer) and consists of a sequence of values, each annotated with (1) a log-specific sequence number that is incremented from 0 with every new value appended to the log, and (2) an incremental cryptographic digest of all log entries up to itself. Only a suffix of the log is stored in A2M, starting with the slot in the "low" position $\mathcal{L} \geq 0$ and ending with the last slot in the "high" position $\mathcal{H} \geq \mathcal{L}$.

A2M essentially offers reliable services a bit-commitment scheme [Nao91] for sequential logs, placed within the trusted computing base. Section 2.3.1 describes the A2M interface, Section 2.3.2 presents simple usage scenarios illustrating how A2M can help a service to remove equivocation from the arsenal of Byzantine-faulty parties, and Section 2.3.3 explores the implementation options for A2M, along with the trust-efficiency trade-off for each.

2.3.1 Interface

An A2M log offers methods to `append` values, to `lookup` values within the log or to obtain the `end` of the log, as well as to `truncate` and to `advance` the log suffix stored in memory. There are no methods to replace values that have already been assigned.

- `append`(q,x) takes a value x, appends it to the log with identifier q, increments the highest assigned sequence number $\mathcal{H}$ by 1, populates the slot at that position with x, and computes the cumulative digest $d_{\mathcal{H}} = h(\mathcal{H} \| x \| d_{\mathcal{H}-1})$, where $d_0 = 0$. This method does not cause any values to be forgotten, i.e., it does not affect $\mathcal{L}$; if the log is unable to allocate storage to the new entry, the method fails.

- `lookup`$(q,n,z) \rightarrow \langle \text{LOOKUP}, q, n, z, x, w, n', d \rangle_{A2M_q,\infty,1}$ takes log identifier q, a sequence number n and a nonce z (for freshness), and returns a LOOKUP attestation. w is the type of the attestation: if sequence number n has not been assigned yet (i.e., $n > \mathcal{H}$) then w is UNASSIGNED and $n' = \mathcal{H}$; if n was assigned once but has now been forgotten (i.e., $n < \mathcal{L}$), then w is FORGOTTEN and $n' = \mathcal{L}$; if slot n has been skipped over via the `advance` method (see below) then w is SKIPPED and n' is the sequence number of the `advance` call that caused the skip; finally, if n is a slot that was filled via `append` or `advance` (see below), then w is ASSIGNED and $n' = n$. x and d are the assigned log value and digest when w is ASSIGNED) and 0 otherwise.

- `end`(q,z) is similar to `lookup`, but returns the last entry of the given log (currently in position $\mathcal{H}$). Attestations from `lookup` and `end` have the same format except for the request name END in the beginning.

- `truncate`(q,n), where $n \in (\mathcal{L}, \mathcal{H}]$, forgets all log entries with sequence numbers lower than n, setting $\mathcal{L}$ to n. All subsequent `lookup` requests for entries below$n$ will be henceforth of type $w =$ FORGOTTEN.

- `advance`(q,n,d,x) allows log q to skip ahead by multiple sequence numbers. It takes a sequence number $n > \mathcal{H}$, a digest d, and a value x. It operates similarly to `append`, but instead of using $d_{\mathcal{H}-1}$ in the digest computation, it uses the given d; skipped sequence numbers are reported as SKIPPED in `lookup`s. Any subsequent `lookup`(q,n'',z)

request for a sequence number n'' that was skipped by this `advance` will return an attestation of the form $\langle \text{LOOKUP}, q, n'', z, x, \text{SKIPPED}, n', d \rangle_{A2M_q,\infty,1}$, which contains information about the `advance` method that caused the skip, until the slot is finally FORGOTTEN.

2.3.2 A2M Usage

Equipped with A2M in its trusted computing base, a reliable service can mitigate the effects of Byzantine faults in its untrusted components, by being able to rely on some small fallback information about individual operations or histories of operations that cannot be tampered with.

During setup, the untrusted component (e.g., a server) must make known to all possible verifiers (e.g., clients or other servers) the authentication keys for its A2M module and the identifier of the A2M log used for each distinct purpose. As far as a verifier is concerned, the A2M authentication key and log identifier are part of the untrusted component's identity. Therefore, a particular A2M-enabled component is allowed to use only its associated A2M.

An untrusted component $\mathcal{C}$ can commit individual data items or operations by `append`ing them to an A2M log. For example, to prove that it has committed to a data item D, the component can execute `append`$(q, h(D))$. The data item is hashed before appending to facilitate A2M implementations in which every log slot has a fixed length.

An interested verifier can establish that the data item D is, indeed, in the untrusted component's committed state by demanding a LOOKUP attestation. To return this attestation, the untrusted must commit $h(D)$ to $A2M_{\mathcal{C}}$. The attestation has a form $\langle \text{LOOKUP}, q, n, z, x, \text{ASSIGNED}, n, d \rangle_{A2M_{\mathcal{C}},\infty,1}$ for some sequence number n and nonce z, where $x = h(D)$ [2]. This conclusively establishes that the untrusted component indeed put the data item D somewhere into its committed log. The sequence number n can be further constrained (e.g., it can be associated with individual protocol steps) to ensure that the untrusted component only commits a single data item for that protocol step; in this sense, multiple verifiers who are mutually disconnected can be assured that the component cannot

[2]Note that we use $A2M_p$ to denote the authentication principal corresponding to host p's A2M module. Trusting A2M means that host p cannot forge authenticators by $A2M_p$ without A2M's cooperation, and that even then, it can only coerce A2M to generate such authenticators as per the A2M interface.

equivocate on the contents of its n-th slot.

To ensure that the untrusted component has a particular data item as the last element in its log, a verifier can provide the untrusted component with a random nonce z and demand the attestation $\langle \textsc{End}, q, n, z, x, \textsc{Assigned}, n, d\rangle_{A2M_C,\infty,1}$. As long as the request type is END, the nonce is the verifier-supplied nonce, and the value $x = h(D)$, the verifier can establish that as of the time of nonce transmission to the component, the last entry in the log was that containing D, and thus no trailing entries were spuriously chopped off by the untrusted component.

The untrusted component is not bound to committing to individual data items in sequential log slots; it can use `advance` to skip some sequence numbers. For example, if it only needs to commit to a value for every k-th sequence number, instead of $\texttt{append}(q, h(D))$ as above, it can use $\texttt{advance}(q, n, 0, h(D))$ for $n = ik$. Invocation of `advance` does not "un-prove" things that the A2M has attested to before. It merely gives up the ability to attest to a real value for the skipped sequence numbers, and disassociates the newly appended request's digest from the log's cumulative history digest thus far, which is not required when committing to individual data items.

When interested in entire histories of data items (e.g., request logs), verifiers can make use of not only the committed data item itself, but also the cumulative digest d. Thanks to the collision-resistant properties of the hash function used, there is a single sequence of data items appended to log q for which the cumulative digest is d. Therefore, by comparing the digests in two LOOKUP attestations from two different untrusted servers, a verifier can establish conclusively that the two servers have committed to the same history up to the looked up sequence number. `advance` can be used, as above, to disassociate two portions of the log, for example, when part of the log is missing during a node's recovery.[3]

To revisit the scenario of a storage server that maintains a log for committed client requests but maliciously drops some off the end when talking to a victim client (Section 2.2.5), consider forcing the server to maintain that log in A2M. Client b can demand a fresh END attestation from the server's A2M log, along with the history itself, and en-

[3]It is important to point out that agreement of two A2M logs on the same sequence number and digest *does not imply* necessarily that the two logs must also agree on attestations about all preceding sequence numbers and digests; the use of `advance` legitimately contradicts this implication. It is possible to change the interface so as to guarantee this implication, but this is not required for our case studies in this thesis.

sure that the included digest is indeed the cumulative digest of the history; this guarantees to *b* that the server has not omitted any requests from the end of its committed log in its response, eliminating this particular problem. Similarly, to revisit the replicated scenario in which malicious replicas profess to different committed requests to different non-faulty replicas, convincing them to commit divergent requests (Section 2.2.5), consider requiring replicas to place such messages into an A2M message log before transmitting them. Now a non-faulty replica, before it allows itself to be convinced by another replica's message, ensures that the message is attested in a LOOKUP attestation drawn from the message sender's A2M message log. In this way, the faulty replica cannot equivocate to two different non-faulty replicas to effect the scenario.

These simple illustrations miss many finer details. We present detailed A2M-enabled protocol designs that achieve fault tolerance that they did not possess before, or increase their fault tolerance, in Sections 2.4 and 2.5.

2.3.3 Implementation Considerations

The fundamental premise behind an implementation of A2M is that it is harder to subvert than the main application. Different implementation scenarios (illustrated in Figure 2.4) lead to different threat models and degrees of trust in the resulting system, and are appropriate for different applications. Our contribution is a novel division of functionality between trusted and untrusted components, not a specific implementation of it – our experimental evaluation in Section 2.6 is a proof of concept, but other implementation scenarios are possible, some of which we characterize below.

The implementation scenarios we present are:

- A separate service offered by a trusted provider or a hardened component (Figure 2.4(a)) that requires a separate machine providing A2M and are slow to access.

- A software-isolated module (Figure 2.4(b)) that requires writing a software-protected lightweight process and is very fast to access since both an A2M-enabled application and A2M are in the same address space.

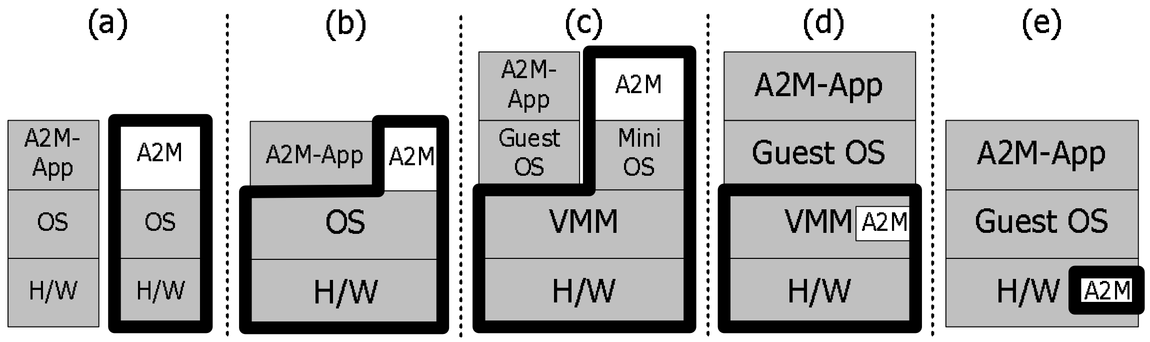

Figure 2.4: A2M implementation scenarios. Thick boxes delineate the trusted computing base. (a) trusted service, (b) trusted software isolation, (c) trusted VM, (d) trusted VMM, and (e) trusted hardware.

- A trusted virtual machine (Figure 2.4(c)) that requires A2M running in a separate virtual machine and is fast to access.
- A trusted virtual machine monitor (Figure 2.4(d)) that requires adding a hypervisor interface to A2M inside the hypervisor and is fast to access.
- Trusted hardware (Figure 2.4(e)) that requires programming a trusted hardware board to implement A2M and can be extremely fast due to hardware cryptographic acceleration.

These implementations are viable in the face of different threats. All five implementations work under the faulty application model (external attacks against server software) but only (a) and (e) work under the faulty operator model (malicious operators that own, operate, and can manipulate entire servers).

In the simplest case, A2M can be a software abstraction implemented as a service visible to applications via an RPC-like interface (Figure 2.4(a)). For instance, it could be a service offered by a trusted provider, such as Amazon's S3 [s3], or by a separate, hardened component with significantly greater assurances in the face of software errors than the main application software and hardware. This is similar to notarization-like approaches [HS91, MB02, YC07] that rely on a trusted write-once medium external to the main system. Though the entire application stack can fail (application, operating system, and hardware), as long as the A2M is running on a trusted system the application can be protected. The big drawback with this implementation scenario is its network-bound nature – in fact, many of its prior instances in practice use this external write-once medium once a day or so – as well as the requirement that everyone needs on-line access to the trusted A2M service provider. Applications with fairly slow request rates such as shared backup services, long-term digital preservation, or certificate authorities may be able to absorb the high-latency interaction with A2M in their relatively infrequent state changes.

Figure 2.4(b) presents a more decentralized approach, in which the A2M implementation relies on the software-based isolation between A2M and an A2M-enabled application. This approach takes advantage of programming language type and memory safety for isolation. Therefore, A2M can be implemented as a library. For instance, in the Singularity [HL07] operating system, the A2M module would be a program that runs as a separate software-isolated process in the same address space. If the Singularity isolation mechanism is trusted, it is possible to trust A2M even if the A2M-enabled application is untrusted. Similarly, in the Java Virtual Machine (JVM) [jav], an application using A2M runs in a sandbox, which constitutes a safe execution environment. The assumption is that if the JVM interpreter, JVM core classes, and an operating system that runs the JVM can be trusted, A2M can be trusted, even if the A2M-enabled Java application is not. Though the isolation is no longer physical as with the scenario of Figure 2.4(a), communication between the application and A2M is fast since they are both in the same address space.

Figure 2.4(c) presents the A2M implementation that relies on the inherent fault isolation properties of a virtual machine monitor (VMM). In the figure, the A2M module is a user-space program running on a small, verifiable operating system on top of a VMM. As long as the VMM and the mini-operating system are trusted to be exploit-free, it is possible to trust the A2M abstraction, even if the application and its general-purpose operating system are compromised. For instance, the virtual Trusted Platform Module (vTPM) [BCG$^+$06] has this architecture. Communication between the application and A2M is only subject to VMM-optimized RPCs, which systems such as Xen [BDF$^+$03] make very efficient.

Further reducing the trusted footprint, the A2M implementation could be placed within the VMM, as in Figure 2.4(d). Here, the assumption is that a small VMM (or, indeed, a microkernel) can be carefully implemented (or formally verified) as bug-free, isolating the correctness of the A2M implementation from potential operating system or application errors above the VMM. For instance, Xen's trusted hypervisor interfaces [BDF$^+$03] could host such an implementation scenario. Both VMM approaches reduce the cost of contacting A2M and can yield efficient, interactive performance for applications such as file systems or transaction processing systems.

Finally, Figure 2.4(e) places the A2M within the hardware itself. Since it tends to be much harder to coerce a hardware module to operate against its specification than it is for software modules, especially without physical access to the hardware, this scenario provides the greatest level of trust in A2M. Hardware implementation options might be to extend a standard Trusted Platform Module (TPM) with some additional non-volatile RAM or an Intel Active Management Technology (AMT) chip [amt], or to use a programmable secure coprocessor such as IBM's commercially available PCIXCC [AD04] board, a programmable PCI-X card with cryptographic primitives as well as physical and electrical tamper-resistance. Tamper resistance offers increased *physical security*: even a malicious host operator armed with electrical probes cannot coerce A2M to give responses that are inconsistent with its specification or to reveal its authentication key material, except for extremely expensive physical cryptanalytic attacks that are unrealistic for most practical situations. Moreover, whereas in the past tamper resistance implied low performance, products such as the PCIXCC coprocessor make a hardware A2M implementation potentially the best performing one – albeit most expensive – among our scenarios. Nevertheless, pervasive hardware implementations of new programming abstractions tend to be slow to arrive, slow to change, and slow to turn into commodities, making this a more tenuous scenario, except for the most sensitive applications.

In this thesis, we experiment with a software A2M implementation. Values stored within A2M logs can have a configurable fixed size, e.g., 32 bytes. The A2M sequence number field needs to have a size large enough to hold sequence numbers of long-running applications (e.g., 160 bits). We implement authentication based on both digital signatures and MACs (with a slightly modified interface from that in Section 2.3.1), though we describe the digital signature version of all protocol designs for simplicity.

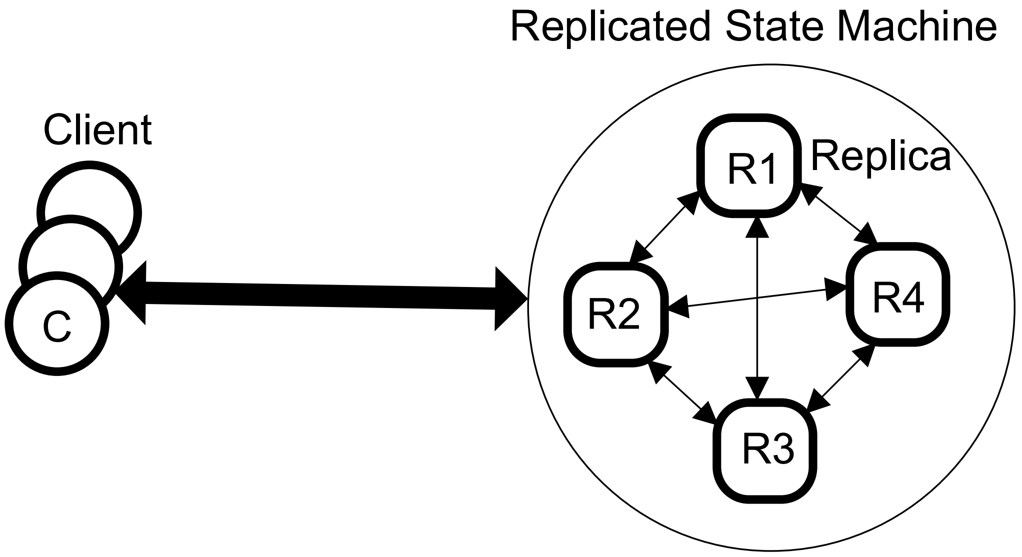

Figure 2.5: Replicated state machine. Clients send requests to servers, servers agree on the sequence of requests to execute, execute requests in the agreed order, and send replies back to the clients.

2.4 A2M State Machine Replication Protocols

In this section, we present state machine replication protocols through the use of A2M, improve their fault tolerance by rendering equivocation extinct or evident. First, in Section 2.4.1, we present a brief overview of the salient features of Castro and Liskov's PBFT protocol for replicated state machines. Second, in Section 2.4.2, we present a simple extension of PBFT, in which A2M protects clients from the replicas' misbehavior, retaining PBFT's safety and liveness for up to $\lfloor \frac{N-1}{3} \rfloor$ faulty replicas out of N, but also guaranteeing safety without liveness for up to $2\lfloor \frac{N-1}{3} \rfloor$ faulty replicas. Second, Section 2.4.3 goes further to protect not only clients from replica misbehavior in PBFT, but also replicas from each other, allowing the fault tolerance of the protocol to go up to $\lfloor \frac{N-1}{2} \rfloor$ with *both* safety and liveness.

2.4.1 Background: PBFT

Castro and Liskov's PBFT protocol [CL02] is a replicated, fault-tolerant mechanism for implementing a *state machine* [Sch90]: an abstraction that represents a deterministic service, in which a starting state (e.g., an empty database) and the sequence of read-compute-write operations at the service determine precisely the state of that service at the end of the operation sequence (Figure 2.5). Such state machines are relatively straightforward to implement on a single, single-threaded server at an individual computer, though any faults at that computer always cause a service failure. For fault-tolerance reasons, it often makes sense to implement the state machine abstraction over a population of such potentially faulty computers interconnected via a potentially faulty network, hoping that even if some computers fail, the service as a whole can continue functioning correctly. Unfortunately, implementing the state machine abstraction over such a population and network is no simple task. In PBFT, each participating computer implements the entire state machine on its local replica of the service state, and replicas communicate with each other to ensure that they all execute the same sequence of operations, and mask individual computers' faults. We describe the protocol in more detail below.

In PBFT, a client c multicasts a request message $\langle \textsc{Request}, o, t, c\rangle_{c,R,1}$ to the N service replicas in replica set R, where o is the operation requested, and t is the timestamp. The client accepts a reply for its request (and only then can submit another) when it receives $\lfloor\frac{N-1}{3}\rfloor+1$ valid matching $\textsc{Reply}$ messages, forming the *reply certificate* $\langle \textsc{Reply}, v, n, t, c, r\rangle_{R,c,\lfloor\frac{N-1}{3}\rfloor+1}$, where v is the view number, n is the assigned sequence number, and r is the result of the request. A *view* is a particular assignment of roles to replicas: the single active *primary* vs. the passive *backups*; when the primary changes, so does the view number v.

Replicas linearize requests via a three-phase agreement protocol (Figure 2.6), starting when the primary (chosen to be the replica with identifier $p \equiv v \mod N$) multicasts to R a newly received request message *req*, encapsulated within a message $\langle \textsc{PrePrepare}, v, n, req\rangle_{p,R,1}$. When backup replica i receives this $\textsc{PrePrepare}$, it multicasts to R a $\langle \textsc{Prepare}, v, n, req\rangle_{i,R,1}$ message. Once replica j has collected $2\lfloor\frac{N-1}{3}\rfloor+1$ $\textsc{PrePrepare}$ or $\textsc{Prepare}$ messages from distinct replicas for this request (which constitute the *prepared certificate* for this request of the form $\langle \textsc{Prepare}, v, n, req\rangle_{R,R,2\lfloor\frac{N-1}{3}\rfloor+1}$), the request becomes *prepared*. To complete the protocol, a replica with a prepared request then multicasts to R a $\langle \textsc{Commit}, v, n, req\rangle_{j,R,1}$ message. When replica k col-

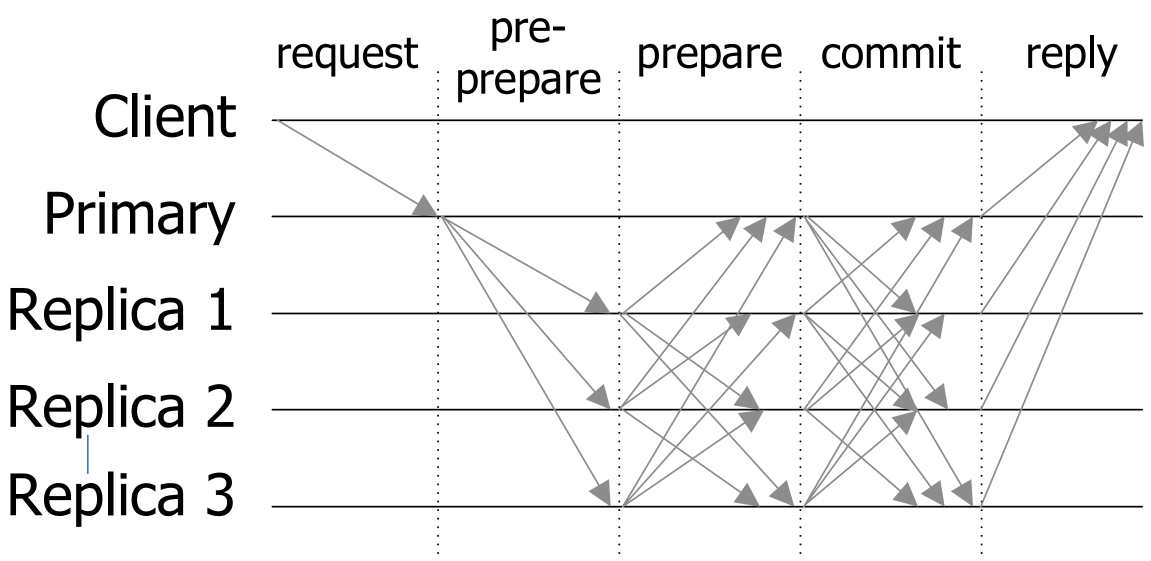

Figure 2.6: Three-phase agreement protocol of PBFT.

lects $2\lfloor\frac{N-1}{3}\rfloor+1$ such messages (which constitute the *committed certificate* of the form $\langle\textsc{Commit}, v, n, req\rangle_{R,R,2\lfloor\frac{N-1}{3}\rfloor+1}$), the replica has established the linearized sequence for this request, committing to execute it as soon as it can; this concludes the *agreement* portion of the PBFT protocol for this request, whose purpose is to ensure that the replicas agree on a single operation sequence for the service, as more clients submit requests for further operations.

A replica can execute the request in its local state as soon as it has finished executing the committed requests for all sequence numbers lower than n. It packages the result in a REPLY message, which it sends to the client directly. When the client has received a quorum of such matching replies – the reply certificate described above – the *execution* portion of the protocol concludes; the purpose of the execution portion is to represent to the client accurately the service state (and reply to the client's request accordingly), as determined by executing the sequence of operations that the agreement protocol portion maintains.

Though the request log can itself represent the service state, replicas periodically garbage-collect their operation log to reduce storage consumption: they create a checkpoint of their local state at a particular sequence number n and a cryptographic hash s of that state. When replica i creates such a checkpoint, it multicasts to R a $\langle\textsc{Checkpoint}, n, s, i\rangle_{i,R,1}$

message. Once it has collected a checkpoint certificate $\langle \textsc{Checkpoint}, n, s\rangle_{R,R,2\lfloor\frac{N-1}{3}\rfloor+1}$, the replica deems that checkpoint "stable," and truncates its operation log up to sequence number n.

When replica i has out-of-date service state (e.g., due to transient network partitions or because it is slow), it can catch up with the rest by retrieving missing committed requests, along with their committed certificates, from another, more up-to-date replica. If other replicas no longer have those certificates in their logs due to garbage collection, the lagging replica can fetch the latest stable checkpoint and certificate, and then any subsequent committed requests after that checkpoint.

Finally, PBFT has a view-change protocol that changes the system's primary when the primary is suspected faulty. When backup replica i in view v times out waiting for a request to commit, it suspects the primary as faulty, and multicasts to R a $\langle \textsc{ViewChange}, v+1, n, s, C, P\rangle_{i,R,1}$ message, where n is the sequence number for the latest stable checkpoint, s is the digest of the stable checkpoint, C is a stable checkpoint certificate, and P is a set of prepared certificates whose sequence number is higher than n.

When a new primary ($p = v+1 \bmod N$) collects a new view certificate V that consists of $2\lfloor\frac{N-1}{3}\rfloor+1$ valid ViewChange messages containing correct C and P, it multicasts to R a $\langle \textsc{NewView}, v+1, V, O\rangle_{p,R,1}$ message, where O is a set of PrePrepare messages in the new view. To determine O, let ℓ be the sequence number of the latest stable checkpoint in V, and let u be the highest sequence number in P. For each sequence number between $\ell+1$ and u, the primary creates a PrePrepare message if a prepared certificate exists in V, or a PrePrepare message for a no-op operation otherwise (to skip that sequence number in the new view).

When a backup replica receives a NewView message, it verifies O is correctly computed by performing the same procedure as the primary. If the message is valid, the replica adds the new information to its log, logs and multicasts to R Prepare messages for each message in O, and enters view $v+1$. The backup processes messages with a view number v' higher than the current view only after it receives a valid NewView message for v'.

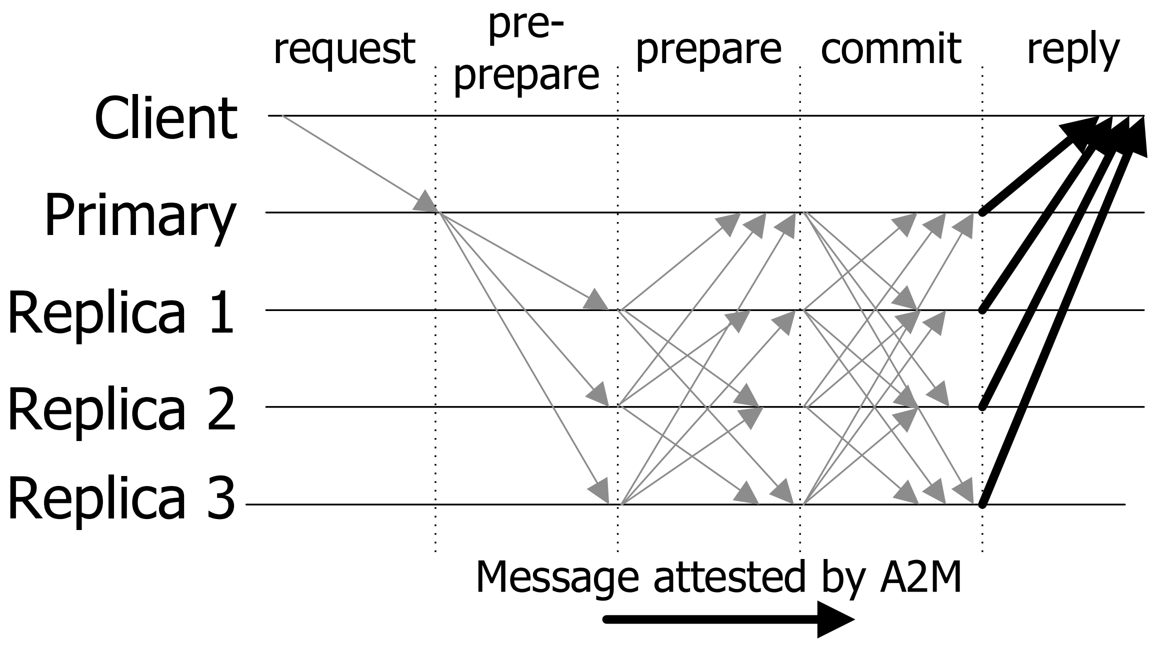

Figure 2.7: Three-phase agreement protocol of A2M-PBFT-E. Thicker lines denote messages that are attested to using A2M.

2.4.2 A2M-PBFT-E

In this section, we describe A2M-PBFT-E, a simple extension of PBFT that uses A2M logs to protect the execution portion of PBFT (hence the “E” suffix of the acronym); that is, it ensures that replicas cannot equivocate about their locally computed results for a particular requested client operation when replying to that or any other client (Figure 2.7). As before, we consider a population R of N replicas.

Design

Replicas: An A2M-PBFT-E replica i maintains all state maintained by a PBFT replica, as well as an A2M log for what it believes as the agreed request sequence; that log has identifier q_i. Other replicas and clients identify this replica as a pair $\langle i, A2M_i \rangle$ of principals, i for the replica node itself, and $A2M_i$ for the replica’s A2M module. As a convenience, we use $A2M_R$ to mean the set of all A2M principals used by replicas in R.

An A2M-PBFT-E replica is functionally identical to a PBFT replica with regards to agreement, but differs on protocol aspects that involve execution, namely client interaction and checkpoint management.

Once replica i collects a committed certificate for sequence number n, it executes the request req on its local application state obtaining result r, it appends the associated request to its log q_i with $\texttt{append}(q_i, h(req))$, and uses $\texttt{lookup}(q_i, n, n)$ to obtain the A2M attestation $\langle \text{LOOKUP}, q_i, n, n, h(req), \text{ASSIGNED}, n, d\rangle_{A2M_i,R,1}$. Finally, it packages the regular PBFT reply message and the attestation into a single message, which it sends back to the client.

As per PBFT, replica i performs garbage collection on its log and A2M request history by exchanging CHECKPOINT messages. When replica i creates a checkpoint, it multicasts to R a $\langle\langle \text{CHECKPOINT}, n, s, d', i\rangle_{i,R,1}, \langle \text{LOOKUP}, q_i, n, n, x, \text{ASSIGNED}, n, d\rangle_{A2M_i,R,1}\rangle$ message where n is the sequence number of the last executed request to produce the checkpoint state, s is the state digest, d' is the A2M digest for sequence $n-1$ (need not be attested), and x is the hash of the n-th committed request. The checkpoint becomes stable when a replica collects a checkpoint certificate $\langle\langle \text{CHECKPOINT}, n, s, d'\rangle_{R,R,2\lfloor\frac{N-1}{3}\rfloor+1}, \langle \text{LOOKUP}, n, n, x, \text{ASSIGNED}, n, d\rangle_{A2M_R,R,2\lfloor\frac{N-1}{3}\rfloor+1}\rangle$. The replica adds this information to its log, removes all messages with sequence number up to n from the log, and performs $\texttt{truncate}(q_i, n)$.

When replica i performs a state transfer, it performs the regular-PBFT process of fetching and installing a state with a stable checkpoint certificate and subsequent agreement messages into its message log. In addition to this, an A2M-PBFT-E replica must also update its A2M request log, by performing $\texttt{advance}(q_i, n, d', x)$, and then $\texttt{append}$ing all subsequently committed requests in ascending sequence order.

Clients: In A2M-PBFT-E, a client c is identical to a PBFT client, except it expects from replica i reply messages of the form $\langle\langle \text{REPLY}, v, n, t, c, r\rangle_{i,c,1}, \langle \text{LOOKUP}, q_i, n, n, h(req), \text{ASSIGNED}, n, d\rangle_{A2M_i,R,1}\rangle$ for its pending request req. This is the PBFT REPLY along with the A2M-attested content of the n-th A2M log entry at the sender. To consider its request completed and accept the result, a client waits until it collects a reply certificate $\langle\langle \text{REPLY}, v, n, t, c, r\rangle_{R,c,2\lfloor\frac{N-1}{3}\rfloor+1}, \langle \text{LOOKUP}, n, n, h(req), \text{ASSIGNED}, n, d\rangle_{A2M_R,R,2\lfloor\frac{N-1}{3}\rfloor+1}\rangle$.

Note that the size of the reply certificate is $2\lfloor\frac{N-1}{3}\rfloor+1$ in A2M-PBFT-E, as opposed to

$\lfloor \frac{N-1}{3} \rfloor + 1$ in PBFT. However, the popular read-only optimization in PBFT – in which read-only requests can be answered by replicas immediately upon reception without a three-phase commit – also requires replies of size $2\lfloor \frac{N-1}{3} \rfloor + 1$, making this difference moot in practice.[4]

Correctness

At a high level, we show that if at most $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty, A2M-PBFT-E does not cause clients to accept more replies than they would under PBFT (therefore does not violate safety) and does not block operations that would have proceeded in PBFT (i.e., does not remove liveness). When the number of faulty replicas ranges between $\lfloor \frac{N-1}{3} \rfloor + 1$ and $2\lfloor \frac{N-1}{3} \rfloor$, we show that A2M-PBFT-E can only assign to any sequence number a unique client request, and that the reply delivered to clients for any sequence number is that which a non-faulty replica would have produced processing the sequence requests in order.

Theorem 1. *If no more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty, A2M-PBFT-E provides both safety and liveness.*

Proof. When no more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty, the safety of A2M-PBFT-E follows from PBFT's safety: A2M-PBFT-E attestations in replies at worst *prevent* a client from accepting a reply that PBFT would otherwise accept (if the REPLY portion of the message matches but the A2M portion does not); A2M-PBFT-E attestations never cause what would have been an unacceptable set of REPLY messages in PBFT to be acceptable. The same holds for liveness, since the addition of the A2M log attestation in REPLY messages cannot hinder progress: there exist at least $2\lfloor \frac{N-1}{3} \rfloor + 1$ non-faulty replicas that maintain their A2M request logs correctly, and as a result, there always exists a quorum of $2\lfloor \frac{N-1}{3} \rfloor + 1$ replicas that can provide clients with a REPLY certificate. Replicas can also create a stable checkpoint since there always exists a quorum of $2\lfloor \frac{N-1}{3} \rfloor + 1$ non-faulty replicas to produce a CHECKPOINT certificate. □

[4]A2M-PBFT-E supports this read-only optimization by replacing LOOKUP attestations with END attestations in the client reply, and using a client-supplied nonce in the attestation, when handling a read-only request; this proves to the client that the result provided is drawn from the latest state of the service, rather than an earlier state (in which case, faulty up-to-date replicas would have advanced their committed request log beyond the attestation they are required to return freshly).

Theorem 2. *If faulty replicas are more than $\lfloor\frac{N-1}{3}\rfloor$ and no more than $2\lfloor\frac{N-1}{3}\rfloor$, A2M-PBFT-E provides safety.*

Proof. When faulty replicas are more than $\lfloor\frac{N-1}{3}\rfloor$ and no more than $2\lfloor\frac{N-1}{3}\rfloor$, we argue inductively that for every sequence number, any non-faulty client can only accept a unique request – which establishes that there exists a single linearized schedule of requests – and can only accept the correct result value for that linearized schedule. In the base case, consider a client accepting req_1 for sequence $n = 1$. Since the corresponding REPLY certificate (of size $2\lfloor\frac{N-1}{3}\rfloor + 1$) includes at least one non-faulty replica, the reply and result certainly correspond to what that non-faulty replica would do with a singleton schedule containing only req_1. Suppose another non-faulty client accepts a different request req_2 and result for the same sequence number $n = 1$. Such a client would also possess a valid REPLY certificate of the same size; the two certificates contain at least one replica in common. However, since that replica is bound by A2M to supply the same A2M log entry to both clients, the A2M attestation of that replica present in the two certificates must be identical, which means that the two certificates must match; this means $req_1 = req_2$, since the request hashes using a collision-resistant hash function also match. This is a contradiction, so there can be no such req_2.

The inductive step for sequence number $n+1$ given a linearized schedule up to n is similar. Any two clients accepting a reply for $n+1$ will have matching requests for that sequence number (as witnessed by the matching request hashes in the two log attestations), *and* matching request histories up to that sequence number (as witnessed by the digest d in the A2M log attestations). Therefore, the result computed by the non-faulty replica in each of the two reply certificates must correspond to the same request history and, due to the deterministic nature of the state machines we consider here, must produce the same result.

Replicas participating in a reply that have used the state transfer mechanism at some point in their history do not affect this correctness argument. After accepting a stable checkpoint certificate, a replica has an n-th A2M log entry that is identical to all the replicas in the checkpoint certificate, including at least another non-faulty replica. Furthermore, the state described in the checkpoint is that held by at least another non-faulty replica. $\square$

Discussion

In the A2M-PBFT-E presentation above, A2M is used to protect only the sequence of committed requests, as they are presented to clients in REPLY messages. However, when faulty replicas are at least $\lfloor \frac{N-1}{3} \rfloor + 1$, they can confuse non-faulty replicas by equivocating during agreement. For example, in Figure 2.2, the use of A2M will not prevent the faulty replicas from causing non-faulty replica r_0 to place request req_a in its A2M position 1 and, at the same time, causing non-faulty replica r_3 to place req_b in its A2M at the same position. Though no client will accept inconsistent replies (since reply messages contain A2M attestations), the replicas themselves are not protected. For the purposes of the protocol, one of the two non-faulty replicas effectively becomes faulty when convinced to adopt a fork in the request history.

The great benefit of A2M-PBFT-E is that such misbehavior causes the system to stop making progress but not to violate its correctness breaking linearizability. In the simplest scenario, an operator who notices lack of forward progress can take the system off-line, identify the history fork (where committed histories diverged), repair the divergent replicas, change their A2M log identifiers, advance their new A2M logs to an earlier correct sequence number from which A2M-PBFT-E can do state transfers, and restart the system with no loss beyond transient unavailability and human effort.

However, a natural next step is to remove this denial-of-service attack from the arsenal of the adversary, by ensuring that the agreement portion of the protocol is itself also protected from equivocation. In the next section, we describe A2M-PBFT-EA, a PBFT extension that protects not only the execution portion (i.e., client-facing messages) against equivocation, but also the agreement portion (i.e., replica-facing messages), thereby increasing the fault tolerance of PBFT with both safety *and* liveness.

2.4.3 A2M-PBFT-EA

To protect against equivocation during agreement, A2M-PBFT-EA (the “EA” suffix stands for *Execution+Agreement*) requires replicas to append to A2M logs all protocol messages before sending them to their peers (Figure 2.8). Unlike the history log, message logs need not protect a sequence of entries, but only an individual message; therefore,

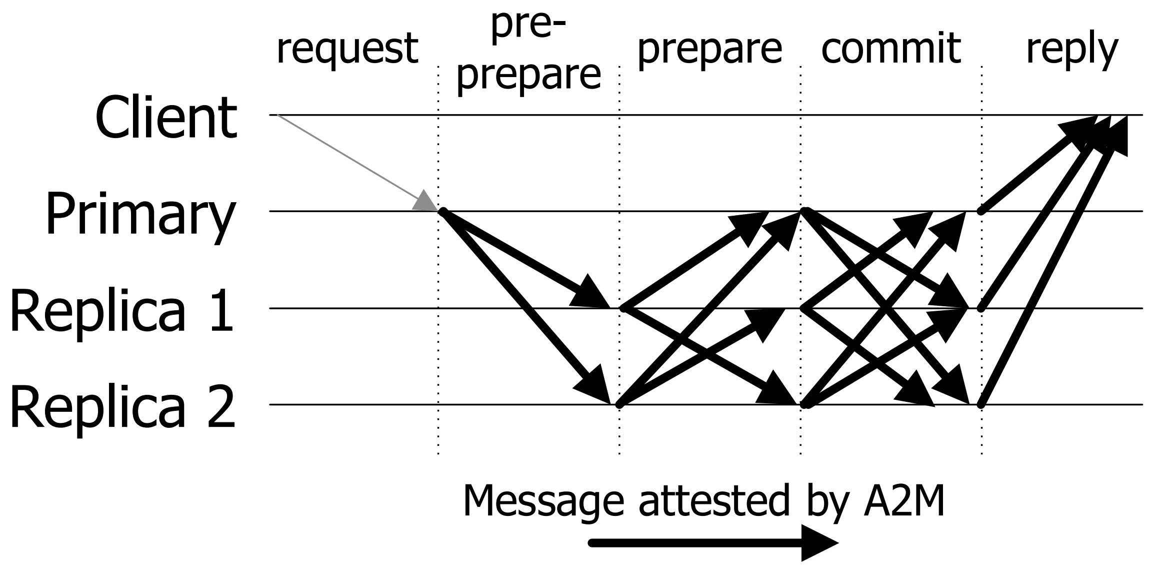

Figure 2.8: Three-phase agreement protocol of A2M-PBFT-EA. Thicker lines denote messages that are attested to using A2M.

A2M's `advance` is used to place a message into an A2M message log, as opposed to `append`. Unlike A2M-PBFT-E and PBFT, which can have multiple requests in flight at the same time, in A2M-PBFT-EA we require that non-faulty replicas handle one request at a time, in increasing sequence-number order.[5] This ensures that messages are appended to their corresponding A2M logs in the order of their corresponding sequence number. By protecting protocol steps from equivocation, A2M-PBFT-EA requires only one – potentially faulty – replica in the intersection of two quorums. Note, in comparison, that PBFT requires at least one *non-faulty* replica in the intersection of two quorums.

When configured with A2M-PBFT-E's quorum sizes, A2M-PBFT-EA has the same safety and liveness properties as A2M-PBFT-E. In what follows, we instead present A2M-PBFT-EA with quorum sizes that allow it to tolerate up to $\lfloor \frac{N-1}{2} \rfloor$ faults with both safety and liveness.

[5]PBFT offers a runtime setting (the high- and low-watermark values) that can be configured to guarantee this requirement.

Design

Clients: An A2M-PBFT-EA client is similar to an A2M-PBFT-E client, but it expects reply certificates of size $\lfloor\frac{N-1}{2}\rfloor+1$ instead of $2\lfloor\frac{N-1}{3}\rfloor+1$.

Replicas: All certificates (for prepared and committed requests, for view changes, and for checkpoints) in A2M-PBFT-EA have size $\lfloor\frac{N-1}{2}\rfloor+1$, as opposed to $2\lfloor\frac{N-1}{3}\rfloor+1$ in A2M-PBFT-E.

In addition to a committed request history log, an A2M-PBFT-EA replica *i* maintains five message logs: PREPARE (which also contains PREPREPARES) and COMMIT for the three-phase agreement, CHECKPOINT for garbage collection, and VIEWCHANGE and NEWVIEW for view changes. Before sending any such PBFT message $\langle\mathcal{M}\rangle$, an A2M-PBFT-EA replica inserts that message to the corresponding message log $m_{\mathcal{M},i}$ (via an `advance` call), uses `lookup` to obtain an attestation $\langle\mathcal{E}\rangle_{A2M_i,R,1}$ for that message, and sends $\langle\langle\mathcal{M}\rangle,\langle\mathcal{E}\rangle_{A2M_i,R,1}\rangle$ to the intended destination. Conveniently, a message that has been committed to A2M in this way need not itself be authenticated to its destination principal; the A2M attestation of the message hash is enough to protect that message from integrity attacks and to make it non-repudiable. Non-attested messages still need to be authenticated as before. Since message logs are typically used for individual attestations and not for message histories, an `advance` call is sufficient, as opposed to an `append`.

A non-faulty replica might have to send multiple versions of a PREPREPARE/PREPARE or a COMMIT message for a given sequence number *n*, but for different views. The protocol *flattens* the $\langle v,n\rangle$ identifier of such messages to fit them in the A2M log entry sequence space, by partitioning log sequence numbers into two parts: the *x* most significant bits (e.g., 64 bits) represent a view number while the remaining *y* bits (e.g., 96 bits) represent a PBFT request sequence number. The log entry number for a PREPREPARE/PREPARE or COMMIT message about view *v* and sequence number *n* is then $n+v2^y$; we use $[v|n]$ to denote this flattened number in what follows. Note that the A2M module is oblivious to this “overloading” of its sequence number space; no changes are required to the A2M interface.

To illustrate the concepts of message attestation and identifier flattening, we present as an example the prepare phase of A2M-PBFT-EA. Where a PBFT replica i would send the PREPARE message $prep = \langle \text{PREPARE}, v, n, req \rangle$, an A2M-PBFT-EA replica commits the message to its corresponding log m_p by invoking `advance`$(m_p, [v|n], 0, h(prep))$, extracts the corresponding LOOKUP attestation $att = \langle \text{LOOKUP}, m_p, [v|n], [v|n], h(prep), \text{ASSIGNED}, [v|n], d' \rangle_{A2M_i,R,1}$, and then bundles and sends $\langle prep, att \rangle$. When an A2M-PBFT-EA replica receives such an attested PREPARE message, it verifies the A2M authentication, and then checks that the value attested is the hash of the included PREPARE message. When a replica collects $\lfloor \frac{N-1}{2} \rfloor + 1$ such messages that match req for the same sequence number n and view v, the request is prepared. The commit phase is similar to the prepare phase described. The checkpoints, state transfer, and execution portions of A2M-PBFT-EA are the same as with A2M-PBFT-E, except for the addition of message attestations in certificates and the different quorum sizes.

View Change: View changes are different from PBFT and A2M-PBFT-E. In PBFT, the quorum forming a NEWVIEW certificate is guaranteed to contain at least one non-faulty replica with the latest committed requests, thanks to the quorum size and the maximum number of faulty replicas. In contrast, the A2M-PBFT-EA quorum size can guarantee, in the worst case, that a single potentially-faulty replica with the latest committed requests will participate in the view change. To address the challenge, an A2M-PBFT-EA replica must be forced to give its latest A2M-committed information, which requires a fresh, shared nonce in the associated `lookup` A2M operations. To accomplish this, the protocol requires an extra phase before the normal view-change protocol, which enables replicas to construct a fresh nonce for the subsequent phases (via WANTVIEWCHANGE messages). For similar reasons, the protocol must ensure that replicas committed to a view change (as evidenced by their issuance of an attested VIEWCHANGE message) cannot subsequently help commit requests in the previous view. Therefore, a VIEWCHANGE message in A2M-PBFT-EA requires the sending replica to explicitly *abandon* the previous view: a replica does this by `advance`ing its COMMIT message log to the end of the old view and attesting to this advancement within its VIEWCHANGE message.

Next, we present the detailed A2M-PBFT-EA view change protocol When replica i in view v_{from} suspects the primary is faulty as per the PBFT protocol, it broadcasts to R its intent to change views via a $\langle \text{WANTVIEWCHANGE}, v_{to}, z, i \rangle_{i,R,1}$ message, where z is a fresh nonce and v_{to} is $v_{from} + 1$ if the replica was not already in the midst of a view change, or

$v+1$ if the replica was in the process of switching to view v when it decided to change yet again.

When a replica collects a WANTVIEWCHANGE certificate that consists of $\lfloor \frac{N-1}{2} \rfloor + 1$ valid WANTVIEWCHANGE messages for the same view v_{to}, it computes the appropriate nonce $\mathcal{Z}$ for its attestations by hashing together all the nonces in its WANTVIEWCHANGE certificate in increasing replica identifier order. It abandons its current view v_{from} if $v_{from} < v_{to}$ (or its participation in a prior view change protocol towards view v' if $v' < v_{to}$), as well as all intervening views up to v_{to}. For all views v in $[v_{from}, v_{to})$ in order, the replica performs $\texttt{advance}(m_c, [v+1|0]-1, 0, 0)$ (if it has not already); $[v+1|0]-1$ is the last COMMIT log entry belonging to view v. Now the replica constructs its VIEWCHANGE message.

The message form is $\langle\langle \text{VIEWCHANGE}, v_{from}, v_{to}, n, s, C, P, Q, W, A, B, H\rangle, \langle \mathcal{E} \rangle_{A2M_i,R,1}\rangle$. Among the contents of the main message, v_{to}, n, s, and C are as in regular PBFT; v_{from} is as defined above, Q is the set of committed certificates with sequence number higher than n and P is the set of prepared certificates for requests that are prepared but are not committed after n, W is a WANTVIEWCHANGE certificate, A is the set of A2M COMMIT log attestations corresponding to the certificates in P, B contains the view abandonment attestations from the replica's COMMIT log (see below), and finally H is a list of committed request log entries that attest those requests in Q. $\langle \mathcal{E} \rangle$ is the attestation from the sender's A2M message log for VIEWCHANGE messages, computed via a $\texttt{lookup}(m_{vc}, v_{to}, v_{to})$ A2M command.

For each abandoned view v between v_{from} and v_{to}, the set B contains the attestation $\langle \text{LOOKUP}, m_c, [v|n'+1], \mathcal{Z}, 0, \text{SKIPPED}, [v+1|0]-1, d'\rangle_{A2M_i,R,1}\rangle$, where m_c is the COMMIT log identifier, and n' is the highest sequence number in Q and P. For each abandoned view, this attestation shows that the replica could not have committed a request for a sequence number greater than those included in its Q and P sets.

When a new primary ($p = v_{to} \bmod N$) collects a new view certificate V that consists of $\lfloor \frac{N-1}{2} \rfloor + 1$ valid VIEWCHANGE messages that have the same v_{from} and v_{to} and contain correct C, P, Q, W, A, B, and H, it multicasts to R a NEWVIEW message of the form $\langle\langle \text{NEWVIEW}, v_{to}, V, O_c, O_p\rangle, \langle \mathcal{E} \rangle_{A2M_p,R,1}\rangle$; the latter part is the usual A2M attestation for the message, whereas the contents of the message are a new view certificate, with the set O_c containing PREPREPARE messages for requests to be committed, and the set O_p containing PREPREPARE messages for requests to be prepared in view v_{to}. When a replica receives the valid NEWVIEW message, it enters view v_{to}. Any requests in prepared or committed certificates for sequence numbers later than the latest stable checkpoint are prepared (issuing a new attested COMMIT message) and committed (appending the request in the request log if not already there) in order, without need for further inter-replica communication.

Note that all VIEWCHANGE messages within a NEWVIEW certificate must have the same v_{from}; this is essential for the correctness properties described next. If the primary fails to collect a quorum of such messages, it refuses to generate a NEWVIEW message. To ensure progress, any non-faulty replica that receives a VIEWCHANGE message with a v_{from} later than its own asks the issuer of that message for the NEWVIEW certificate that allowed it to enter v_{from}. Using that certificate, the lagging replica can bring itself to that view. When a timeout indicates that the previous view change attempt stalled – either due to a faulty new primary or because of v_{from} mismatches – the replica initiates another view change for the next target view number. Thanks to the eventual synchrony of our network, this guarantees that eventually enough replicas will initiate a view change with the same v_{from} and the change will go through.

Correctness

At a high level, A2M-PBFT-E and A2M-PBFT-EA differ in two fundamental ways: on one hand A2M-PBFT-EA has smaller quorum sizes, but on the other hand, it requires all protocol messages to be attested to from an appropriate A2M log before use.

Theorem 3. *If no more than $\lfloor\frac{N-1}{2}\rfloor$ replicas are faulty, A2M-PBFT-EA provides safety.*

Proof. The argument presented in the second proof of Section 2.4.2 also applies to the safety of A2M-PBFT-EA. Therefore, it guarantees safety with up to $\lfloor\frac{N-1}{2}\rfloor$ faults since clients accept REPLY certificates of size $\lfloor\frac{N-1}{2}\rfloor+1$. □

To show that A2M-PBFT-EA is live despite up to $\lfloor\frac{N-1}{2}\rfloor$ faults, we show a new safety invariant that is not necessary for linearizability: all non-faulty replicas agree on a single committed request sequence. That is, a faulty replica cannot convince two non-faulty replicas to commit to their respective A2M request logs different requests for the same sequence number. The argument is split into a same-view case and a different-view case. For the same-view case, it follows backwards the agreement process from appending a request to the log, to emitting a COMMIT message, to emitting a PREPARE message, showing that for two different requests to be placed in two non-faulty replicas' request logs, some A2M must be faulty, which is incompatible with our fault model. For the different-view case, the argument is similar, but must also traverse NEWVIEW certificates; view abandonment in such certificates helps show that it is not possible for a single replica (faulty or not) to have an attested COMMIT message for one request in one view, and at the same time support a view change feigning ignorance for that message, leading to a contradiction.

We prove that if no more than $\lfloor \frac{N-1}{2} \rfloor$ replicas are faulty, A2M-PBFT-EA provides liveness by showing that no two non-faulty replicas can place different requests in the same sequence number of the A2M request log. We split our argument into a same-view case, and a different-view case.

Theorem 4. *If no more than $\lfloor \frac{N-1}{2} \rfloor$ replicas are faulty, no two non-faulty replicas can place different requests in the same sequence number of the A2M request log in the same view.*

Proof. Suppose two non-faulty replicas have appended two different requests to the same sequence number of their respective A2M request logs, during the same view. They both did that after having constructed a valid committed certificate over two quorums. Those two quorums must have at least one common (perhaps faulty) replica i, which managed to attest to two COMMIT messages, one for each request, in each of the two quorums. This, however, is a contradiction with our assumption that A2M is trusted to avoid equivocation for the same log entry, and the collision-resistance of the hash function.

It is worth noting that along similar lines, it is trivial to show that no two non-faulty replicas can be convinced to place different requests in their COMMIT A2M log for the same sequence number and view, by the analogous argument on the prepared certificate quorums and the PREPARE A2M log of the common replica. Finally, the exact same argument can be used to show that no two non-faulty replicas can put different requests in their PREPARE A2M logs for the same sequence number and view, since the single primary for the view can only attest to a single PREPREPARE message for that sequence number in any given view. $\square$

Theorem 5. *If no more than $\lfloor \frac{N-1}{2} \rfloor$ replicas are faulty, no two non-faulty replicas can place* *different requests in the same sequence number of the A2M request log across different views.*

Proof. Now we must show that no two non-faulty replicas can commit two requests r and $r' \neq r$ in sequence n and in views v and $v' > v$, respectively.

We define an *active* view as a view for which a valid NEWVIEW certificate has been constructed *and* seen by a non-faulty replica. A non-faulty replica cannot commit a request in a view for which it has not seen a valid NEWVIEW certificate, therefore if a non-faulty replica commits a request in a view, then that view must be active.

We split our argument into two further subcases, first the case in which no other active views exist between v and v', and the case in which at least one active view exists between v and v'.

Case 1 – v and v' are consecutive active views: Since no other active views exist between v and v', then the NEWVIEW certificate for v' – and there can be at most one since only one NEWVIEW message can be attested by the primary for view $v_{to} = v'$ – must have $v_{from} \leq v$. This is because at least one non-faulty replica must have produced a VIEWCHANGE message for the certificate, and that non-faulty replica guarantees that its v_{from} represents an active view, which cannot be later than v (or it would have to be v'). As a result, this NEWVIEW certificate contains view abandonments for all views in its $[v_{from}, v_{to})$ range, which includes $[v, v')$ as we argued above.

Now consider three quorums, the one that produced the committed certificate for r in view v (denoted $\mathcal{Q}$), the one that produced the NEWVIEW certificate to v' (denoted $\mathcal{V}$), and the one that produced the committed certificate for r' in view v' (denoted $\mathcal{Q}'$). Let $i \in \mathcal{Q} \cap \mathcal{V}$, which always exists thanks to quorum intersection.

Replica i unavoidably contributed an attested COMMIT message for r at sequence number n in the committed certificate for v along with the rest of quorum $\mathcal{Q}$. What can have been i's VIEWCHANGE contribution to the NEWVIEW certificate in quorum $\mathcal{V}$ with regards to sequence number n? If i reported a valid stable checkpoint no earlier than n in its VIEWCHANGE, then the resulting, unique NEWVIEW certificate for v' should convince any non-faulty replica that sees it to never commit anything else at n in view v', since n belongs in the past; this contradicts our assumption that some non-faulty replica will in fact commit

r at n in view v'.

If instead i reported a stable checkpoint earlier than n in its VIEWCHANGE, it can only have reported the same COMMIT attestation for request r at n, since that VIEWCHANGE message must contain a view abandonment for v as we showed above, and omitting an attestation for the COMMIT log entry $[v|n]$ is not an option; to omit it successfully, it would have to produce an abandonment attestation $\langle \text{LOOKUP}, m_c, [v|n'+1], \mathcal{Z}, 0, \text{SKIPPED}, [v+1|0]-1, d' \rangle_{A2M_i,R,1}$ for some $n' < n$, which is disallowed by the A2M interface given the existence of an ASSIGNED attestation for entry $[v|n]$ and the inequality $[v|n'+1] \leq [v|n] < [v+1|0]-1$.

This leaves the common replica i between quorums $\mathcal{Q}$ and $\mathcal{V}$ only with the option of reporting request r as prepared in view v. As a result, any correct replica in quorum $\mathcal{Q}'$, which can only commit requests in view v' after having seen the NEWVIEW certificate for that view, must have issued at least a PREPARE message for request r in view v' while processing the NEWVIEW certificate. However, since this replica is also a member of the committed certificate for request r' in view v', it must also have prepared and subsequently committed that request r'. This clearly contradicts not only the properties of the A2M message logs at that replica, but also the operation of a non-faulty replica. This completes the proof for this subcase.

Case 2 – v and v' are not consecutive active views: Suppose there are $v_1, v_2, \ldots, v_{k-1}$ active views between $v(= v_0)$ and $v'(= v_k)$. We can prove inductively on the intervening active views that at least a prepared certificate for request r at sequence n will be propagated to view v', preventing a commitment of a conflicting request r' at the same sequence number there.

In the base case, we can use the argument of the previous subcase 1 to show that the NEWVIEW certificate for view v_1 will either preclude any subsequent commitment to sequence n or will contain at least a prepared certificate for request r at that sequence number.

To show the inductive step, assume that the NEWVIEW certificate for view v_i contains a prepared certificate for request r – that is the only viable choice since, if it contains a stable checkpoint for n or later, then no subsequent view will admit a different committed request r', leading to a contradiction. Now consider the NEWVIEW certificate, formed by quorum $\mathcal{V}$, that will lead away from v_i to v_{i+1}. Any non-faulty replica in $\mathcal{V}$ (there must be at least one), must have seen the earlier NEWVIEW certificate leading to v_i, or else it would be unable to assume v_i as its active view. Therefore, that replica must also have prepared that same request r in view v_i, including the prepared certificate in its VIEWCHANGE contribution to the later NEWVIEW certificate.

The induction proves that committed request r at n in active view v will either preclude the commitment of another request at n in view v' (because somewhere in between a NEWVIEW certificate contained a stable checkpoint for a sequence at or after n), or cause the inclusion of a COMMIT attestation for the same r at n in all subsequent valid NEWVIEW certificates. This contradicts the assumption that a non-faulty replica at active view v', which must have seen such a NEWVIEW certificate, will commit request r' at n in view v'. This last subcase concludes the proof that two commitments for the same sequence number at different non-faulty replicas must commit the same request. $\square$

Beyond quorum availability (i.e., ensuring that no quorum can be blocked from forming due to non-faulty replicas caused to commit incorrect requests), A2M-PBFT-EA also guarantees that no replica is left behind during view changes: a replica only abandons its current view v if it has collected a WANTVIEWCHANGE certificate; even if the current view change does not complete due to network faults or a faulty new primary, the replica can retransmit the WANTVIEWCHANGE certificate until eventually enough other non-faulty replicas have received it to complete the view change, or to trigger another one with a different primary. This is guaranteed by the eventual synchrony of our network and processing model.

2.5 Other A2M Protocols

In this section, we describe A2M-Storage, an A2M-enabled storage system on a single untrusted server shared by multiple clients. Thanks to the use of a trusted A2M module, A2M-Storage provides linearizability in contrast to SUNDR's weaker fork consistency and is simpler than SUNDR. We then briefly sketch how A2M can be used with Q/U to improve its fault tolerance.

2.5.1 A2M-Storage

Background: SUNDR

SUNDR targets the same problem as PBFT: linearize client requests and ensure that the service state used to respond to each request corresponds to a correct system having executed this linear request history. In PBFT, agreement is used among replicas to obtain a linearized request order. The presence of at least one non-faulty replica corroborating a reply to the client ensures that the agreed upon linearized order has been executed correctly producing the result in the reply. Unfortunately, in a single-server environment such as SUNDR's, there is no non-faulty replica trusted to execute linearized requests; instead, the clients must trust each other and cooperate to check themselves that requests are properly linearized and execution is performed correctly at the server.

A SUNDR server maintains the current service state (a snapshot of a shared file system), which is represented by Merkle trees [Mer87].[6] The state is captured by a set of version structures, each of which is owned by a client (principal) and contains a hash that summarizes the whole state on which the client operates.

To perform an operation (read/write on a file), a SUNDR client submits to the server its intended request, called an *update certificate*. The server assigns an order to the request relative to pending operations that have not committed yet, and returns the latest committed version structures and ordered pending update certificates. The client ensures that the state transits correctly forward from its last committed version the server gives via a sequence of pending operations. The client can then perform its operation locally, potentially fetching missing blocks by following digests of the hash tree, compute and sign a new state digest creating a new version structure, and return it along with changed blocks to the server. The server stores the new version structure and modified blocks.

As described in simpler terms in Section 2.2.5, a SUNDR client cannot ensure that the server sends it the latest state resulting from the committed history of requests; though it cannot remove requests from the middle, the server can still chop off the tail of history past the last request known to that client, and start a new "fork" in that history, specific

[6]We omit the details of how files and directories are organized. What is important is that an entire file system can be cryptographically digested and verified against a set of digests efficiently.

to the client. Until two clients on different history forks compare their notes, they cannot know the system is not linearized. This is what makes SUNDR only fork-consistent but not linearizable.

Design

A2M-Storage can be simpler than SUNDR, and guarantees linearizability instead of only fork consistency, thanks to the use of the trusted A2M module, which affords clients the ability to demand the latest committed request on a history, via a fresh END attestation.

The server maintains a version block, a snapshot of a file system captured by a Merkle tree, and two A2M logs. A version block holds a state digest (i.e., the root hash of a snapshot) computed as for SUNDR and a sequence number that tracks the latest A2M log sequence number with a signature signed by the latest writer. A2M has log q_h for the write request history, and log q_s for digests of version blocks, one for each state version generated by the application of writes to the state. Each write/read request is associated with a logical timestamp, of the form $\langle seq, att_{h,seq}, att_{s,seq} \rangle$, containing the request sequence number, the A2M attestation from the request history log q_h when that request was appended, and the A2M attestation from the state version log q_s when that request was executed. The client remembers the latest timestamp it has seen.

An A2M-Storage client performs write operations optimistically, assuming the timestamp it knows is the latest. When it submits a write request *req* for sequence number *n*, it also submits a nonce (for freshness), its known timestamp on which *req* is conditioned, and a new version block with sequence number *n* obtained after executing *req*. If the conditioned-on timestamp has not changed, the server modifies the state accordingly, stores the new version block that the client sends, and appends the request and state version digests to A2M logs q_h and q_s, respectively. In other words, execution of the request is conditioned on the latest timestamp at the server being the same as that known by the client. The server then forms its response, containing a success code, END attestations from the two logs, and a proof that the operation was committed to the service state using the state digest function. The client accepts the response if the attestations and stage digest proofs are valid. If however the client had a stale timestamp, indicated by a failure code in the response, it updates its timestamp with the one returned by the server, and tries again potentially after fetching fresher state blocks and potentially backing off in case of write contention.

An A2M-Storage client performs read operations that include nonces. The server returns END attestations from the two A2M logs whose freshness is proven by a nonce, the version block to which the last A2M q_s entry points, and a proof that the read content is the valid part of the current snapshot. Note that the version block should include the same sequence number as the A2M attestation sequence number to be valid.

Instead of the optimistic, one-phase version of the protocol, a pessimistic two-phase version is straightforward as well, in which clients always fetch a "grant" to perform their operation at a particular sequence number, and then submit their operation with a guarantee of success, as per SUNDR.

In terms of its software architecture, A2M-Storage is similar to a version of SUNDR that entrusts the task of ordering requests and maintaining version structures to a separate, trusted component called a consistency server. In A2M-Storage, this task is "emulated" with the help of A2M, a general-purpose abstraction that works not only for SUNDR but also for other systems as we have demonstrated in other sections.

Correctness

A2M-Storage clients and server need maintain far less state than is necessary for SUNDR: clients only require a single global timestamp, instead of per-client version structures. Yet, A2M-Storage provides linearizability, because a client accepts a write operation as complete only when the server proves that the request is committed to its A2M logs – and A2M logs are trusted not to violate linearization. Similarly, a client accepts a read operation response as complete only when the response carries the latest timestamp, whose freshness is attested by the A2M module.

Theorem 6. *A2M-Storage provides linearizability.*

Proof. We show informally that there exists a sequential history of accepted writes, and that each read is partially ordered to the correct immediately preceding write. When a write operation is accepted by a client, we know that the operation is committed to A2M right after the conditioned-on timestamp. By following a chain of conditioned-on timestamps backwards, we can construct a single history of accepted client write operations. In addition, when a read is accepted by a client, we know that the read response carries the latest committed state version. The read operation can be placed right after the write that produces a state version attested by A2M and on which the read depends. Therefore, there exists a linearizable history of accepted write and read operations. □

Since there is only one server, there is no guarantee on liveness when the server fails. Moreover, due to the nature of optimistic protocols, A2M-Storage does not provide any guarantees on fairness among clients; a greedy client can overuse the system.

2.5.2 A2M-Q/U

The Query/Update protocol (Q/U) [AEMGG+05] is a quorum-based BFT replicated state machine. It offers an optimistic protocol that completes client requests in a single round-trip message exchange between a client and the replicas, in the absence of faults and write contention. At a very high level, Q/U is similar to A2M-Storage (with more than a single server): the client sends a request along with its view of all replicas' latest timestamps, each of which contains a replica's history. Each replica commits the request if its local timestamp is compatible with the client's view; otherwise, e.g., if another client has already advanced that replica's state with another conflicting update request, the replica refuses to execute the request and sends back its latest replica history. A client is satisfied about its request's linearization if a quorum of replicas ($4f+1$ out of $5f+1$ total replicas) accept its request, making it *complete*. If fewer than $2f+1$ replicas have accepted the client's request, then it is *incomplete* and the client tries again after some back-off. When a client receives matching replies from between $2f+1$ and $4f$ replicas, the request is *repairable*. A client attempts to repair a repairable request, by trying to see if enough other replicas exist to make it complete, or by trying to convince other replicas to accept it. If a client's operation is complete, the protocol guarantees that, in any other quorum in the system, that operation would be repairable, a fundamental invariant for Q/U's linearizability guarantee.

Q/U's linearizability properties stem from the sizes of the population N, quorums Q, and repairable sets R, given the number f of tolerable faults. A quorum must be always available even if all faulty replicas remain silent – implying $N \geq Q + f$ (1) – all quorums must intersect over a repairable set, excluding all faulty replicas – implying $2Q - N \geq R + f$ (2) – and all quorums must intersect over at least one non-faulty replica of all repairable sets of other quorums – implying $Q + R - N > f$ (3).

A2M's contribution to Q/U is that, by having replicas place accepted requests into A2M logs and having clients require an END attestation before accepting a replica's response, the sizes of quorum and repairable set intersections can be reduced. Essentially, $f + 1$ replicas form a repairable set since faulty replicas commit to one history with A2M and they cannot form a repairable set with non-faulty replicas with an old history. Therefore, the above condition (3) changes to $Q + R - N \geq 1$. An A2M-enabled Q/U protocol can tolerate f faults with $N = 4f + 1$, $Q = 3f + 1$, and $R = f + 1$, reducing the replication factor required from 5 to 4. We defer the full details.

2.6 Evaluation

In this section, we evaluate the overhead of applying A2M to BFT state machine replication. We have implemented A2M-PBFT-E and A2M-PBFT-EA (without its view change algorithm) in C/C++ with a BFT library [CL02, RCL01] ported to Fedora Core 6 and the SFSlite library [sfs]. The A2M protocols have versions that use signatures or MACs for authentication.

We ran our experiments with four replica nodes for A2M-PBFT-E and one client node. For A2M-PBFT-EA experiments, we use three replica nodes to tolerate one fault. The replica nodes are 1.8GHz Pentium 4 machines and the client node is a 3.2GHz Pentium 4 machine. All machines are equipped with 1GB RAM and 3Com 3C905C Ethernet cards, and are connected over a dual speed 10/100Mbps 3Com switch.

A2M uses SHA-1 as its digest function (also used for MACs), and NTT's ESIGN with 2048-bit keys for signatures. On a 1.8GHz machine, signature creation and verification of 20 bytes take on average $256\mu s$ and $194\mu s$, respectively.

All experiments used A2M as a library in the same address space as the PBFT proto-

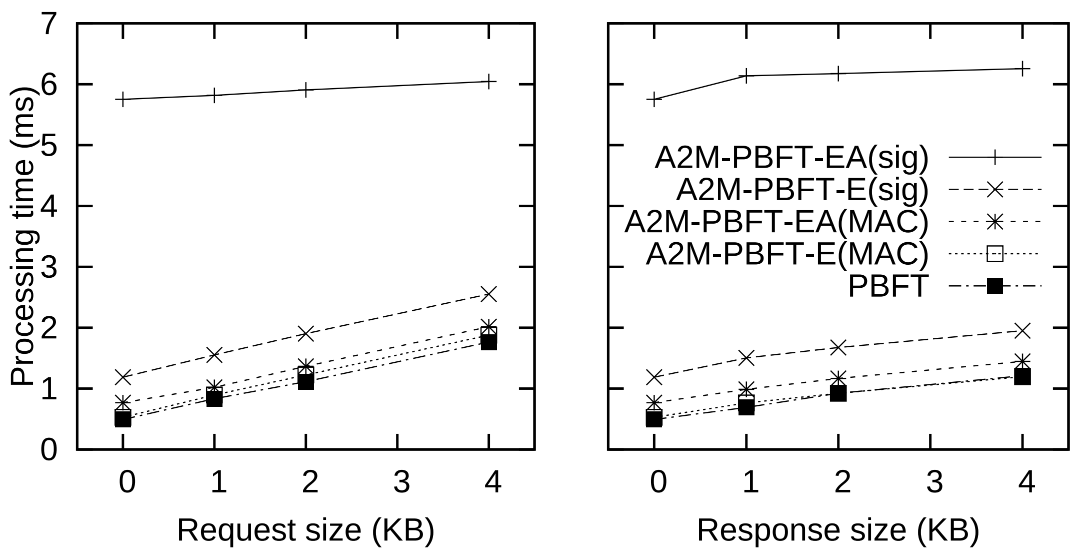

Figure 2.9: Microbenchmark results varying request (left) and response (right) sizes, measured in KBytes. "sig" refers to use of signatures while "MAC" refers to use of MACs in the protocols.

col and the user application. However, depending on the A2M implementation scenario (see Section 2.3.3), A2M operations will experience a different additional interface latency cost. To account for the costs in accessing A2M, we impose by default $1\mu s$ of delay, which is a conservative system call latency[7] (Figure 2.4(d)) or a cross-SIP communication latency [HAF$^+$07] (Figure 2.4(b)), to each A2M request using the Pentium RDTSC instruction.

In our experiments, we compare PBFT to A2M-PBFT-E and A2M-PBFT-EA, using two A2M implementations: one using signatures for authentication (denoted "sig") and one using MACs (denoted "MAC"). Shown PBFT measurements used MACs.

2.6.1 Microbenchmarks

We use a simple microbenchmark program, which is a part of the PBFT library. A simple client sends 100,000 null operation requests of size *a* bytes to replicas, which elicit

[7]On a 1.8GHz Pentium 4 machine running Fedora Core 6, we ran lmbench [MS96] to measure the time to perform nontrivial entry into the operating system. The system call takes 0.87 μs in average.

NFS Phase	-S	-PBFT	-A2M -PBFT-E (sig)	-A2M -PBFT-E (MAC)	-A2M -PBFT-EA (sig)	-A2M -PBFT-EA (MAC)
Copy	0.219	0.709	1.026	0.728	2.141	0.763
Uncompress	1.015	3.027	4.378	3.103	8.601	3.236
Untar	2.322	4.448	6.826	4.553	12.896	4.669
Configure	12.748	12.412	19.173	12.659	26.181	13.040
Make	7.241	7.461	9.778	7.500	11.379	7.510
Clean	0.180	0.298	0.640	0.312	0.742	0.311
Total	23.725	28.355	41.821	28.854	61.940	29.528

Figure 2.10: Mean time to complete the six macro-benchmark phases in seconds.

replies of size *b* bytes from replicas. We ran experiments with *a*'s and *b*'s varying between 0 and 4000. Figure 2.9 plots the results. In all cases, operation turn-around times grow at the same pace with request/response sizes as in PBFT, with an additive overhead due to the additional A2M authentication operations (MACs or signatures) required. A2M-PBFT-E (MAC) and A2M-PBFT-EA (MAC) add a small extra cost because of the relative efficiency of MAC computation compared to the network delays. The signature-based versions of the protocol add significant computational overheads, and only become justifiable for very large replica populations, in which the cost of carrying MAC-based authenticators becomes comparatively expensive.

2.6.2 Macrobenchmarks: NFS

To understand the implications of using A2M-enabled protocols in real applications, we use PBFT's NFS front end on a PBFT (or A2M protocol) back end. As with BFS [CL02], we use a local NFS loop-back server and an NFS kernel client at the client side.

The workload we use consists of compiling a software package (`nano-2.0.3.tar.gz`) in six phases: 1) copy the file to the NFS file system (*copy*), 2) uncompress the file (*uncompress*), 3) untar the uncompressed file (*untar*), 4) run a configure script (*configure*), 5) compile the package by running make (*make*), and 6) clean up the built object and execution files (*clean*). The workload includes 8790 read-only BFT operations out of a total of 14500 operations invoked.

NFS- Additional latency (μs)	A2M-PBFT-E (MAC)	A2M-PBFT-E (MAC) with batching	A2M-PBFT-EA (MAC)	A2M-PBFT-EA (MAC) with batching
1	28.854	28.763	29.528	29.505
10	29.598	29.025	31.299	30.188
50	32.735	30.232	36.242	32.214
250	48.784	37.237	66.441	45.199
1000	117.59	65.813	192.53	101.62

Figure 2.11: Mean time to complete the six macrobenchmark phases in seconds for different A2M additional latency costs.

We compare six NFS-*X* protocols, where *X* is the name of the back-end protocol implementing the NFS interface. In addition to PBFT and our four A2M-enabled variants, we also run NFS-S, which uses a single server without replication. Figure 2.10 shows the average time to complete each phase, out of 10 runs. The standard deviations of all results are within 4% of the mean. NFS-PBFT is 19.5% slower than NFS-S. NFS-A2M-PBFT-E (MAC) and NFS-A2M-PBFT-EA (MAC) are 1.8% and 4.1% slower than NFS-PBFT, respectively, whereas NFS-A2M-PBFT-E (sig) and NFS-A2M-PBFT-EA (sig) are 47.5% and 118.4% slower than NFS-PBFT, respectively. Overall, NFS-A2M-PBFT-E (MAC) and NFS-A2M-PBFT-EA (MAC) achieve significantly better fault tolerance at a slight increase in cost over PBFT.

2.6.3 Effects of A2M Placement

To explore the associated costs of other A2M implementation scenarios, we impose delays to each A2M request, varying delay duration from 10μs (for the order of magnitude of typical inter-process communication) to 1*ms* (for the order of magnitude of RPC on the same LAN).

Figure 2.11 shows the average time to complete the macrobenchmark, out of 10 runs when the additional A2M interface latencies are 10, 50, 250, and 1000μs. The mean times of NFS-A2M-PBFT-E (MAC) are 2.6, 13.5, 68.0, and 307.5% slower than the base NFS-A2M-PBFT-E (MAC) with 1μs delay; the slowdown corresponds to two delayed A2M operations and three A2M MAC verifications per BFT operation. For NFS-A2M-PBFT-

EA (MAC), the mean times are 6.0, 22.7, 125.0, 552.0% slower than the base NFS-A2M-PBFT-EA (MAC) with 1μs delay; the slowdown is greater because of the greater number of A2M operations invoked during agreement steps.

To amortize the effect of this A2M access latency, we explore a multiple-operation batching optimization. In A2M-PBFT-E replicas bundle an `append` with its subsequent `lookup` when they send replies. In A2M-PBFT-EA replicas also bundle an `advance` with their subsequent `lookup` during agreement steps. Furthermore, the client batches A2M MAC verifications. When additional latencies are 1 and 10 μs, this batching effect is negligible. However, when additional latencies are 50, 250, and 1000μs, A2M-PBFT-E with batching improves mean times by 7.6, 23.5, and 44.0% respectively and A2M-PBFT-EA with batching improves mean times by 11.1, 32.0, and 47.2% respectively.

2.7 The Right Abstraction

In the previous sections, we have argued and experimentally demonstrated that systems incorporating in their design a small, trusted abstraction, A2M in our examples, can improve their fault tolerance at certainly tolerable cost. However, an interesting open question remains: is A2M the *right* trusted abstraction, for the types of applications we demonstrated here – state machines, replicated or centralized? Furthermore, is it the right trusted abstraction for other reliable applications that are more loosely organized than replicated state machines?

In systems that strive for linearizability, such as those forming the focus of our work here, the notion of a common event (i.e., request) history is central. Therefore, being able to commit to and compare histories seems, at a minimum, a required trusted function, which is exactly what A2M's log abstraction offers. Arguably, when histories need not be compared, as is the case when ensuring A2M-PBFT-EA replicas commit to their messages before sending them, it is sufficient to be able to commit to individual key-value pairs that are independent of all others, which is a narrower specification than what A2M offers. However, given that the difference between attested key-value pairs and attested logs is small (the computation of an incremental digest with every append), we opted to make a trusted log the basic, common abstraction that covers both replicated and single-server systems.

Would a larger trusted abstraction be preferable? Arguably, one could push an entire replicated state machine protocol, such as PBFT, into the trusted computing base. The application interface exported – an invocation method, and an execution callback [CL02] – is certainly simple, and applies to any deterministic application state machine. For example, one could imagine a trusted implementation of a fail-stop replicated state machine protocol, such as Paxos [Lam98]. However, a replicated state machine abstraction, even one that is trusted not to be Byzantine, remains fairly complex to implement; it requires transmission and reception of network messages and several sets of local variables per request per remote replica. In contrast, A2M requires no network interactions, and only a circular buffer that tends to be short; although a hardware implementation of A2M appears trivial, a hardware implementation of Paxos might not be.

Beyond linearizable replicated state machines, an interesting question might be what other, orthogonal, trusted abstractions might make sense under different consistency requirements. For instance, when dispensing session guarantees weaker than linearizability (such as "read your writes" [PST$^+$97] or fork consistency [LKMS04]), simple trusted logical clocks [Lam78] might be sufficient compared to an abstraction such as A2M.

2.8 Future Work

Although A2M is fruitfully applicable to all shared-state protocols we know of, we hope to investigate other trusted abstractions, such as Lamport clocks and version vectors, and their translation to practical system facilities to further reduce the footprint of the trusted computing base for applications with weaker consistency requirements.

In this thesis, we implemented A2M in a library. We hope to explore other implementation scenarios such as VMM and trusted hardware. We hope to implement a Xen A2M driver for applications running on top of Xen. In addition, we hope to explore the cost of adding A2M to a commercial TPM-like environment.

2.9 Summary

In this chapter, we present a trusted, log-based abstraction called Attested Append-Only Memory (A2M). Servers utilizing A2M are forced to commit to a single, monotonically increasing sequence of operations. Since the sequence is externally verifiable, malicious servers cannot present different sequences to different parties. We discuss several implementation scenarios of A2M under different threat models. We present A2M-PBFT-E, a simple variant of Castro and Liskov's PBFT protocol that can achieve safety with up to $2\lfloor \frac{N-1}{3} \rfloor$ faulty replicas. We also present A2M-PBFT-EA, a more involved variant, that can preserve safety and liveness with up to $\lfloor \frac{N-1}{2} \rfloor$ faulty replicas. Finally, we show how to achieve linearizability in single-server storage systems such as SUNDR. Our prototype implementations of A2M-PBFT-E and A2M-PBFT-EA show minor performance overhead; they are 1.8% and 4.1% slower than the PBFT base case, respectively. There are many technical details in this chapter, but the bottom line is that A2M is a practical and eminently implementable tool for improving the fundamental Byzantine fault tolerance of replicated and centralized systems alike.

Chapter 3

TimeMachine: Long-term Fault Tolerance

3.1 Overview

In this chapter, we investigate long-term Byzantine fault tolerance in the context of digital preservation systems. Digital preservation systems aim to maintain authentic copies of data objects for long periods of time. Such systems face two major challenges: *durability* and *long-term authenticity*. Durability means that preserved data are not lost. Authenticity means that a data item retrieved from the system in the future given an item name is the same as what the original data creator stored into the system under that name in the past.

Traditional systems achieve these attributes via directories inside file systems (authenticity), disks for storage (availability), and tapes for backup (durability) under benign fault assumptions (as illustrated in Figure 3.1). A centralized file server maintains directories that maintain mappings between human-readable names and i-node numbers, disks store data and metadata blocks, and tapes are used for scheduled backups. However, recent trends are that disks replace tapes for maintaining durability [GCB$^+$02, emc] since disks are easier to manage, are readily accessible and highly available, and are easier to cope with failures through automatic replication.

In recent years, researchers have made great progress in distributed storage systems that operate under Byzantine fault assumptions. Self-verifying bitstore systems such as

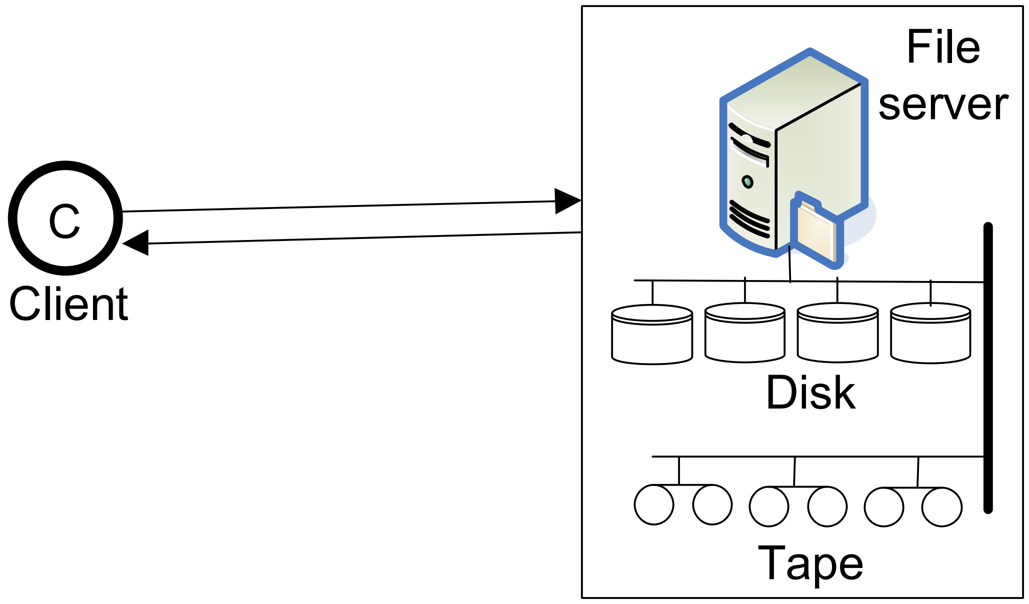

Figure 3.1: Traditional storage system architecture. File system directories are used for authenticity and disks and tapes are used for durability.

OceanStore [KBC+00], PAST [RD01], and Glacier [HMD05] have addressed durability comprehensively, but authenticity has less satisfying solution. They maintain *self-verifying data*, for which the name of a data item is an *authenticator* for that data item, which can be used to verify the item itself (e.g., a cryptographic hash). If there is at least one correct copy throughout the lifetime of the systems, durability is maintained. Often the systems organize data objects of a publisher as a Merkle tree [Mer87] structure where the root block of the tree is signed by the publisher's private key; the tree structure is named by the publisher's public key. Users who can remember such a name (a long string of otherwise meaningless digits) can ascertain long-term authenticity of the corresponding content fetched from a preservation service. This solution does not, however, deal with usage models in which a user decades down the road wishes to authenticate the contents of a preserved document or a publisher's collection of documents by a human-readable name (e.g., "State Budget Fiscal Year 2003", "UCB EECS Snapshot 2002-02-07"). Unfortunately, existing systems provide no solution to preserving the mapping between a human readable name and an authenticator for a data item or a collection of data items. The nature of such fallback authentication information is typically left out of the scope of proposed designs, though in

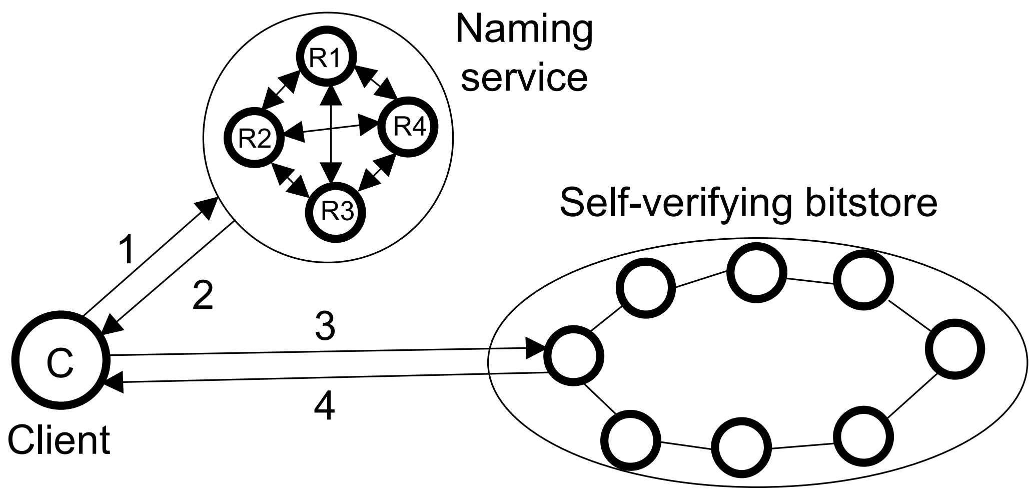

Figure 3.2: Architecture of our data storage, which separates authenticity management from durability management.

its turn it requires long-term preservation as well without the benefit of a further fallback.

In this chapter, we take an approach to separating authenticity management from durability management (Figure 3.2). Typical data objects (e.g., documents, audio/video files, and a collection of files) are significantly larger than metadata objects. Thus, in our architecture data objects are maintained by self-verifying bitstore to reduce replication costs. For authenticity we present a separate trustworthy naming service that preserves mappings between human-readable names and authenticators, which are not self-verifying; thus, we fill the important missing piece of previous archival storage systems.

Before presenting our naming service, we discuss maintaining replicas in a self-verifying bitstore for durability in Section 3.3. We use a standard replica maintenance process that maintains a certain number of replicas by repairing lost replicas. A key question is how to set this replication threshold. We model the process as a continuous time Markov chain and analyze the process to compute an appropriate replication threshold in an operation environment.

Next, we move to our main contribution of this chapter. We close the gap between human readable names and authenticators, in the form of self-certified data names or other cryptographic constructs, by constructing a Byzantine-fault tolerant (BFT) name-to-authenticator mapping service. We argue that existing Byzantine-fault models – which require a bounded number of faulty nodes at *all* times – and correctness properties – which guarantee correct service as long as fault bounds are *never* violated – are inappropriate in the long-term (Section 3.4). Instead, long-term services such as our preserved name service should offer a stronger property called *Healthy-Write-Implies-Correct-Read* (HWICR) we propose; they should guarantee the correct preservation of data added while the system was "healthy" (e.g., fault thresholds were not violated), despite subsequent periods when fault thresholds were violated; a system that goes through a single "unhealthy" period should not be damned to failure forever.

Motivated by the need for a preserved name service and the impracticality of existing BFT approaches in a long-term context, we make three contributions in this domain. First, we advocate a *tiered Byzantine-fault model*, in which bug exploits and similar causes of faulty behavior are *unbounded*, whereas higher-trust components, such as trusted hardware that fails when an entire corporation fails, are held under a tighter fault bound. We argue that the tiered Byzantine-fault model maps well to reality and accommodates well the implications of long-term reliability for data preservation.

Second, we advocate the convergence of proactive recovery and trusted hardware for preservation applications that operate in the tiered Byzantine-fault model. We borrow from proactive recovery systems the distinction between *service phases*, which might suffer bug-related faults, and *proactive recovery phases*, during which the damage done by bug exploits is flushed while smaller, easier-to-verify trusted functionality audits and cleans up system state. From trusted hardware, we borrow the notion of short-term, tamper-resistant functionality for digital signing and small amounts of storage, which we use to justify the bounded-fault tier of our fault model.

Third, in Section 3.5, we outline the use of these facilities in the design of *TimeMachine* (TM), a preserved name service that provides the HWICR property. TM relies on a simple, affordable, and easy-to-build extension to commodity trusted hardware, which allows the service to store a short but sensitive summary of its state where transient faults cannot corrupt it. TM operates in alternating phases of service and proactive recovery. In contrast to traditional proactive recovery, TM places operations that change its state within the recovery phase, leaving for its vulnerable service phases only operations that read existing state and whose correctness can be certified using the trusted hardware. TM provides correct results regardless of service-phase faults and as long as no more than a third of its trusted hardware devices fail within a single recovery phase; it also guarantees durability if at least one server is non-faulty between consecutive service phases.

Thankfully, preservation applications, where state change can be slow as long as authenticity is guaranteed, fit this structure well. Other applications where state changes can be batched and committed at a relatively low frequency, such as those that tolerate weak consistency for instance, would also be able to benefit from our contributions. However, not all applications can fit this structure, especially those for which state changes need immediate, interactive confirmation such as file systems.

We evaluate our prototype TM implementation to validate our design in Section 3.7, discuss future work in Section 3.8, and summarize in Section 3.9.

3.2 Separating Authenticity from Durability

Self-verifying bitstore systems such as OceanStore [KBC$^+$00], PAST [RD01], and Glacier [HMD05] work well if a client knows the name (e.g., SHA-256 hash) of a data object or the name of a collection of data objects (e.g., publisher's public key). But how will people find out the name, which is an authenticator, in the first place? As an analogy, in file systems, a semantic-free inode number can be used to retrieve a file, but it is not reasonable to expect people to remember the inode number, thus a file name is used to retrieve the inode number and then to retrieve the file. This becomes more important in preservation systems. It is even more challenging for people to remember authenticators for long periods of time, but human-readable names (especially that follow well-defined naming conventions) are easy to remember. Therefore, we argue that we must maintain information that binds human-readable names to data objects (or collections of data objects).

A naive approach is to store mappings between human-readable names and data objects (or collections of data objects). However, since these mappings are not self-verifying, we have to rely on voting to ensure correctness. A quorum of replicas must be maintained correctly to return correct data objects to client requests. Compared to a self-verifying data store, this approach is inefficient since data objects are typically large and we need to replicate more data objects to mask the same number of faults. In particular, to tolerate f faults, a self-verifying data object requires $f+1$ replicas, but a non-self-verifying data object requires $3f+1$ replicas if we use Byzantine-fault tolerant state machine replication.

We take an approach to separating authenticity management from data object durability management. We maintain mappings between human-readable names and authenticators in a naming service for authenticity and store data objects in a self-verifying bitstore. The naming service maintains mappings, each of which is small, using a more costly algorithm to ensure correctness. However, the bitstore maintains self-verifying data objects, which can be organized as a Merkle tree structure. In comparison, to tolerate f faults while maintaining a data object, this approach requires $f+1$ data object replicas and $3f+1$ non-self-verifying metadata object (mapping) replicas, which is cost-effective.

Figure 3.2 shows our storage architecture that separates a naming service for data authenticity from a self-verifying bitstore for data durability. In the architecture, a client retrieves data by following procedures: 1) a client sends a query message with a human-readable name to the naming service, 2) the naming service returns an authenticator (e.g., SHA-256 hash), 3) the client sends a fetch message with this authenticator to a server that stores relevant data, and 4) the server returns the data to the client. We discuss maintaining replicas in a self-verifying bitstore focusing on durability in Section 3.3 and discuss our naming service from Section 3.4.

3.3 Maintaining Self-verifying Replicas for Durability

We study a standard replication algorithm that reacts to replica failures. The algorithm monitors how many replicas are available and it creates new replica(s) if the number of available replicas is below the replication threshold (r_L). The challenge here is to determine the correct replication threshold that is suitable for a given operating environment. Another challenge is to efficiently maintain replicas without spending bandwidth unnecessarily. This is hard since it is not possible to distinguish transient failures (e.g., node reboots) with permanent failures (e.g., disk failures). Techniques to mitigate the effects of transient failures are discussed in Carbonite [CDH$^+$06]. Here we focus on the first challenge.

We consider the problem of providing durability for a storage system composed of a large number of nodes, each contributing disk space. The system stores a large number of independent pieces of data. Each piece of data is immutable and self-verifying. While parts of the system will suffer temporary failures, such as network partitions or power failures, the focus of this section is on failures that result in permanent loss of data.

3.3.1 Challenges to Durability

It is useful to view permanent disk and node failures as having an average rate and a degree of burstiness. To provide high durability, a system must be able to cope with both.

In order to handle some average rate of failure, a high-durability system must have the ability to create new replicas of objects faster than replicas are destroyed. Whether the system can do so depends on the per-node network access link speed, the number of nodes (and hence access links) that help perform each repair, and the amount of data stored on each failed node. When a node *n* fails, the other nodes holding replicas of the objects stored on *n* must generate replacements: objects will remain durable if there is sufficient bandwidth available on average for the lost replicas to be recreated. For example, in a symmetric system each node must have sufficient bandwidth to copy the equivalent of all data it stores to other nodes during its lifetime.

If nodes are unable to keep pace with the average failure rate, no replication policy can prevent objects from being lost. These systems are *infeasible*. If the system is infeasible, it will eventually "adapt" to the failure rate by discarding objects until it becomes feasible to store the remaining amount of data. A system designer may not have control over access link speeds and the amount of data to be stored; fortunately, choice of object placement can improve the speed that a system can create new replicas.

If the creation rate is only slightly above the average failure rate, then a burst of failures may destroy all of an object's replicas before a new replica can be made; a subsequent lull in failures below the average rate will not help replace replicas if no replicas remain. For our purposes, these failures are *simultaneous*: they occur closer together in time than the time required to create new replicas of the data that was stored on the failed disk. Simultaneous

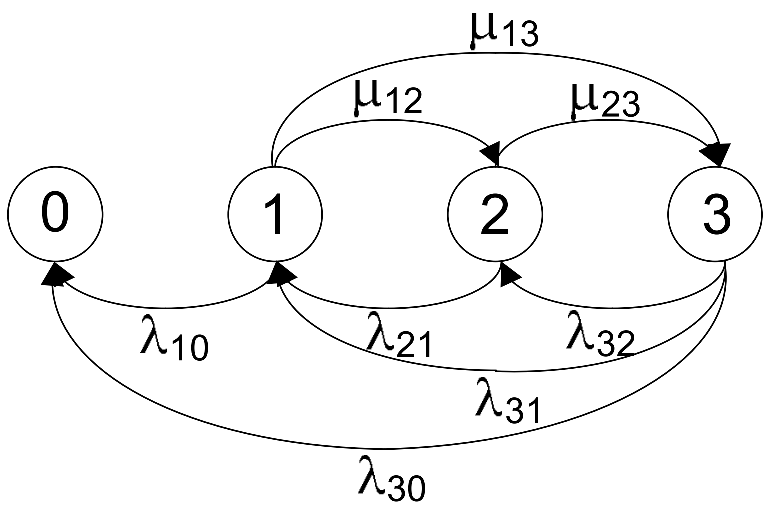

Figure 3.3: A continuous time Markov model for the process of replica failure and repair for a system that maintains three replicas ($r_L = 3$). Numbered states correspond to the number of replicas of each object that are durable. Transitions to the left mean replicas are lost; transitions to the right mean replicas are created.

failures pose a constraint tighter than just meeting the average failure rate: every object must have more replicas than the largest expected burst of failures. We study systems that aim to maintain a target number of replicas in order to survive bursts of failure; we call this target r_L.

Higher values of r_L do *not* allow the system to survive a higher average failure rate. For examples, if failures were to arrive at fixed intervals, then either $r_L = 2$ would always be sufficient, or no amount of replication would ensure durability. If $r_L = 2$ is sufficient, there will always be time to create a new replica of the objects on the most recently failed disk before their remaining replicas fail. If creating new replicas takes longer than the average time between failures, no fixed replication level will make the system feasible; setting a replication level higher than two would only increase the number of bytes each node must copy in response to failures, which is already infeasible at $r_L = 2$.

3.3.2 Creation versus Failure Rate

We model the replica maintenance process as a continuous time Markov chain (CTMC). Figure 3.3 shows this model for the case where $r_L = 3$. An object is in state i when i disks

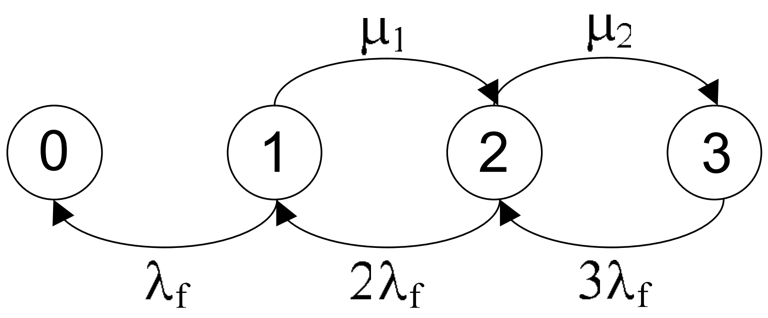

Figure 3.4: A continuous time Markov model for the process of replica failure and repair for a system that maintains three replicas ($r_L = 3$). Numbered states correspond to the number of replicas of each object that are durable. Transitions to the left occur at the rate at which replicas are lost; right-moving transitions happen at the replica creation rate.

hold a replica of the object. There are thus $r_L + 1$ possible states, as we start with r_L replicas and only create new replicas in response to failures. From a given state i, there is a transition to state j with rate μ_{ij} corresponding to repair, except for state 0 which corresponds to loss of durability and state r_L which does not need repair. The actual rate μ_{ij} depends on how bandwidth is allocated to repair and may change depending on the replication level of an object. There is a transition to the state j with rate λ_{ij} corresponding to replica failure.

Under the assumption that independent exponential inter-failure and inter-repair times, which is reasonable in a PlanetLab-like environment, we can simply use a birth-death process (Figure 3.4). From a given state i, there is a transition to state $i+1$ with rate μ_i corresponding to repair. There is a transition to the next lower state $i-1$ with rate $i\lambda_f$ because each of the i nodes holding an existing replica might fail.

This model can be analyzed numerically to shed light on the impact of r_L on the probability of data loss; we will show this in Section 3.3.3. However, to gain some intuition about the relationship between creation and failure rates and the impact this has on the number of replicas that can be supported, we consider a simplification of Figure 3.4 that uses a fixed μ but repairs constantly, even allowing for transitions out of state 0. While these changes make the model less realistic, they turn the model into an M/M/∞ queue [Kle75] where the “arrival rate” is the repair rate and the “service rate” is the per-replica failure rate. The “number of busy servers” is the number of replicas: the more replicas an object has, the more probable it is that one of them will fail.

This simplification allows us to estimate the equilibrium number of replicas: it is μ/λ_f. Given μ and λ_f, a system cannot expect to support more than this number of replicas. For example, if the system must handle coincidental bursts of five failures, it must be able to support at least six replicas and hence the replica creation rate must be at least 6 times higher than the average replica failure rate. We will refer to μ/λ_f as θ. Choices for r_L are effectively limited by θ. It is not the case that durability increases continuously with r_L; rather, when using $r_L > \theta$, the system provides the best durability it can, given its resource constraints. Higher values of θ decrease the time it takes to repair an object, and thus the 'window of vulnerability' during which additional failures can cause the object to be destroyed.

To get an idea of a real-world value of θ, we estimate μ and λ_f from the historical failure record for disks on PlanetLab [CDH+06]. The average disk failure inter-arrival time for the entire test bed is 39.85 hours. On average, there were 490 nodes in the system, so we can estimate the mean time between failures for a single disk as $490 \cdot 39.85$ hours or 2.23 years. This translates to $\lambda_f \approx 0.439$ disk failures per year.

The replica creation rate μ depends on the achievable network throughput per node, as well as the amount of data that each node has to store (including replication). PlanetLab currently limits the available network bandwidth to 150 KB/s per node, and if we assume that the system stores 500 GB of unique data per node with $r_L = 3$ replicas each, then each of the 490 nodes stores 1.5 TB. This means that one node's data can be recreated in 121 days, or approximately three times per year. This yields $\mu \approx 3$ disk copies per year.

In a system with these characteristics, we can estimate $\theta = \mu/\lambda_f \approx 6.85$, though the actual value is likely to be lower. Note that this ratio represents the equilibrium number of *disks* worth of data that can be supported; if a disk is lost, all replicas on that disk are lost. When viewed in terms of disk failures and copies, θ depends on the value of r_L: as r_L increases, the total amount of data stored per disk (assuming available capacity) increases proportionally and reduces μ. If $\lambda_f = \mu$, the system can in fact maintain r_L replicas of each object.

To show the impact of θ, we ran an experiment with the synthetic trace (i.e., with 632 nodes, a failure rate of $\lambda_f = 1$ per year and a storage load of 1 TB), varying the available bandwidth per node. In this case, 100 B/s corresponds to $\theta = 1.81/r_L$. Figure 3.5

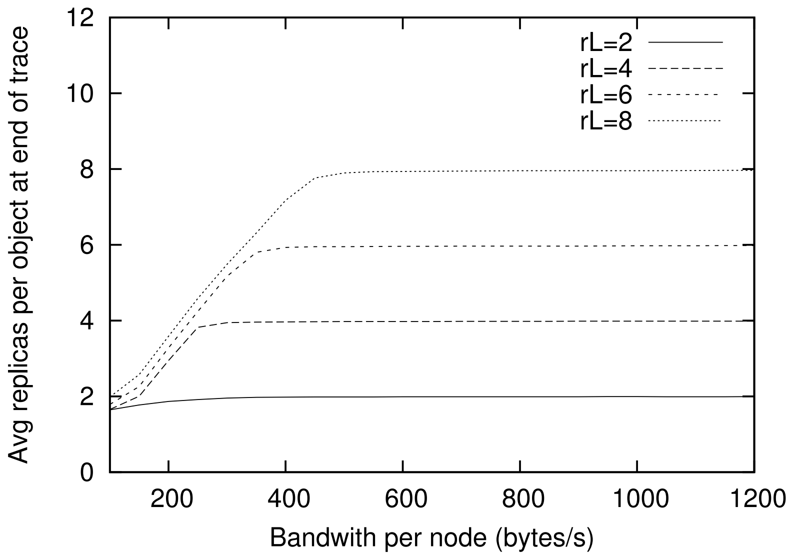

Figure 3.5: Average number of replicas per object at the end of a two-year synthetic trace for varying values of θ, which varies with bandwidth per node (on the *x*-axis) and total data stored (r_L). Where $\theta < 1$, the system cannot maintain the full replication level; increasing r_L further does not have any effect.

shows that, as θ drops below one, the system can no longer maintain full replication and starts operating in a 'best effort' mode, where higher values of r_L do not give any benefit. The exception is if some of the initial r_L replicas survive through the entire trace, which explains the small differences on the left side of the graph.

3.3.3 Choosing r_L

A system designer must choose an appropriate value of r_L to meet a target level of durability. That is, for a given deployment environment, r_L must be high enough so that a burst of r_L failures is sufficiently rare.

One approach is to set r_L to one more than the maximum burst of simultaneous failures in a trace of a real system. For example, Figure 3.6 shows the burstiness of permanent failures in the PlanetLab trace by counting the number of times that a given number of failures occurs in disjoint 24 hour and 72 hour periods. If the size of a failure burst exceeds the number of replicas, some objects may be lost. From this, one might conclude that 12

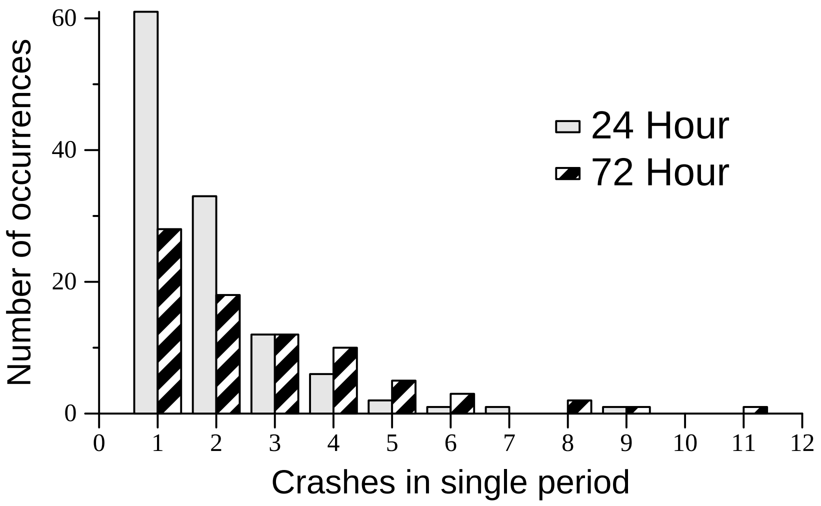

Figure 3.6: Frequency of “simultaneous” failures in the PlanetLab trace. These counts are derived from breaking the trace into non-overlapping 24 and 72 hour periods and noting the number of permanent failures that occur in each period. If there are *x* replicas of an object, there were *y* chances in the trace for the object to be lost; this would happen if the remaining replicas were not able to respond quickly enough to create new replicas of the object.

replicas are needed to maintain the desired durability. This value would likely provide durability but at a high cost. If a lower value of r_L would suffice, the bandwidth spent maintaining the extra replicas would be wasted.

There are several factors to consider in choosing r_L to provide a certain level of durability. First, even if failures are independent, there is a non-zero (though small) probability for every burst size up to the total number of nodes. Second, a burst may arrive while there are fewer than r_L replicas. One could conclude from these properties that the highest possible value of r_L is desirable. On the other hand, the simultaneous failure of even a large fraction of nodes may not destroy any objects, depending on how the system places replicas. Also, the workload may change over time, affecting μ and thus θ.

The continuous time Markov model described in Figure 3.4 reflects the distributions of both burst size and object replication level. The effect of these distributions is significant. An analysis of the governing differential equations can be used to derive the probability that an object will be at a given replication level after a given amount of time. In particular, we can determine the probability that the chain is in state 0, corresponding to a loss of

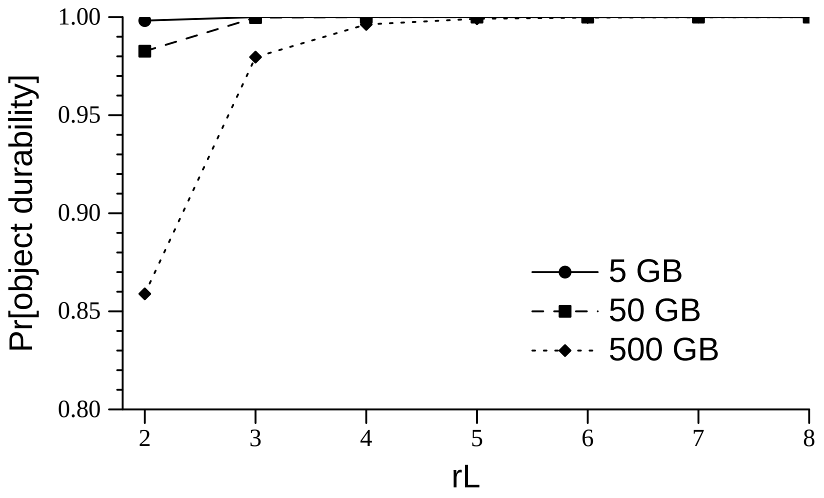

Figure 3.7: Analytic prediction for object durability after four years on PlanetLab. The *x*-axis shows the initial number of replicas for each object: as the number of replicas is increased, object durability also increases. Each curve plots a different per-node storage load; as load increases, it takes longer to copy objects after a failure and it is more likely that objects will be lost due to simultaneous failures.

durability.

We show the results of such an analysis in Figure 3.7. To explore different workloads, we consider different amounts of data per node. The graph shows the probability that an object will survive after four years as a function of r_L and data stored per node (which affects the repair rate and hence θ).

As r_L increases, the system can tolerate more simultaneous failures and objects are more likely to survive. The probability of object loss at $r_L = 1$ corresponds to using no replication. This value is the same for all curves since it depends only on the lifetime of a disk; no new replicas can be created once the only replica of the object is lost. To store 50 GB durably, the system must use an r_L of at least 3. As the total amount of data increases, the r_L required to attain a given survival probability also increases. Experiments confirm that data is lost on the PlanetLab trace only when maintaining fewer than three replicas.

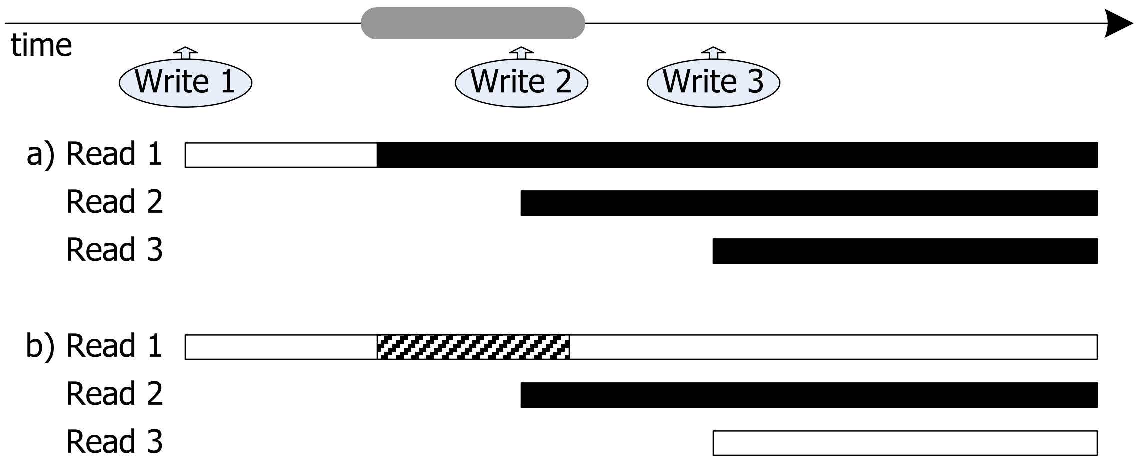

Figure 3.8: Behavior of preservation system under faults. Thick gray portion of timeline indicates period during which fault assumptions are violated. Three ovals indicate additions to the state of the system (new name/value mappings). Horizontal boxes indicate how state additions are seen by client reads – white for correctly/black for incorrectly. Case (a). Typical BFT behavior: once fault bound is violated, no correctness is guaranteed for retrieval of bindings added *before, during, or after* the violation. Case (b). System with HWICR property: Additions that occur *before* or *after* the violation can be correctly observed after the violation is repaired. During the violation period, system guarantees not to return incorrect value for read 1, but may lose availability, i.e., may not return a value at all (shaded box). Addition 2 is lost in both cases, since it occurs during the unhealthy period.

3.4 Towards a Long-term Fault Tolerant Naming Service

In the previous section, we discuss how to maintain replicas for durability in a self-verifying bitstore. Next, we delve into a naming service for long-term data authenticity. We first introduce a new service property and a new, more realistic, Byzantine fault model that are suitable for long-term services.

3.4.1 New Service Property

A Byzantine-faulty node can behave arbitrarily, which includes crashing, or even following a concerted plan with other faulty nodes towards an unknown malicious goal. Typical Byzantine-fault models allow the network to drop, duplicate, and reorder messages, though usually it is assumed that enough retransmissions eventually deliver a message to its destination within some unknown time bound. These models require that the number of faulty nodes does not exceed a hard upper bound, such as a third of the entire population of nodes. This has been justified with the reasoning that in a reliable replicated service, participating nodes are well enough managed, secured, and maintained that they can mostly avoid network-triggered exploits of unpatched bugs and the physical manipulation of unfettered malicious humans. In such a setting, it appears reasonable to provide the usual correctness property: as long as the system is *healthy* – that is, no more than f out of $3f+1$ nodes are faulty at any point in time – the system will offer its correctness guarantees [ZSR02,CL02].

Unfortunately, in long-running systems, uninterrupted good health is tough to guarantee. First, malicious attacks such as virus and worm infections are increasingly hard to stop, even in well managed enterprise settings; the fact that most nodes in a replicated system will be running one or perhaps two distinct implementations and operating systems, prone to the same exploits, does not help the situation either. Furthermore, after decades of continual use, human errors, organizational slip-ups, and other unlikely events are bound to crop up [BSR$^+$06], causing bound violations to occur. Even if one such slip into an *unhealthy period* occurs, the correctness of typical BFT systems can no longer be guaranteed, not just for the duration of the violation, but also forever into the future (Figure 3.8(a)). For example, in a system such as Castro and Liskov's Practical BFT (PBFT) [CL02], once the fault bound is violated, faulty nodes can cause non-faulty nodes to execute distinct, divergent sequences of operations on their local states, from which they cannot recover without human intervention [LM07].

In this chapter, we introduce a stronger guarantee on BFT services called Healthy-Write-Implies-Correct-Read (HWICR) (see Figure 3.8(b)). HWICR requires that the system can be viewed as mapping names to values; a name is uniquely mapped to a value.

Definition 1. *A system provides HWICR iff for every (name, value) mapping stored during a healthy period, the system is guaranteed to return the same value when queried with its name at all future times (or a notice that the value is unavailable), in spite of intervening unhealthy periods.*

A system provides a stronger HWICR property iff it provides HWICR, and for a (name, value) mapping stored during an unhealthy period, the system is guaranteed not to return

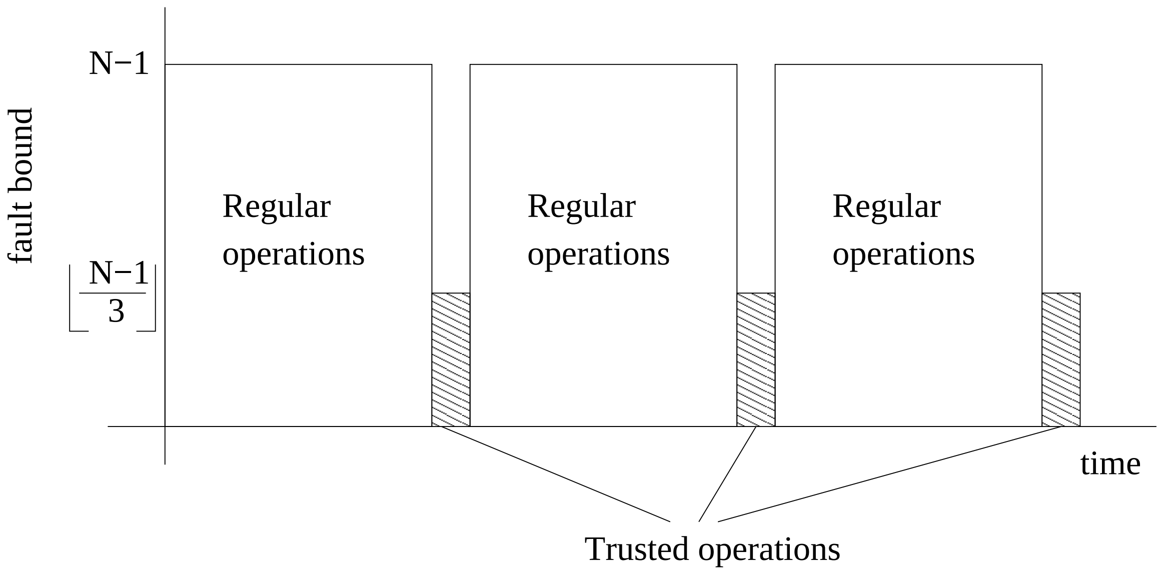

Figure 3.9: Tiered Byzantine-fault model.

an incorrect value when queried with its name at all future times. This is the case when a client's write request is authenticated by its signature. Since faulty replicas in the system cannot forge the signature of the client even during an unhealthy period, forged writes will not be accepted by the system.

HWICR implies two sub-properties, integrity and durability. Integrity means the system never returns an incorrect value when queried with its name and cannot convince clients that a (name, value) mapping does not exist once it is correctly stored. Durability means the system does not lose a (name, value) mapping once it is correctly stored.

3.4.2 The Tiered Byzantine-Fault Model

To achieve HWICR in long-running services, we propose a new fault model, which we call a tiered Byzantine fault model. We assume two types of operations, *regular* and *trusted* operation, with different fault bounds (Figure 3.9). During regular operation periods, there are at most $N-1$ faulty replicas out of N total replicas, i.e., there is at least one correct replica during regular operation periods, but no more than $\lfloor \frac{N-1}{3} \rfloor (= f)$ replicas are faulty during any trusted operation period. As we will see, these assumptions are necessary to achieve the HWICR property. Integrity requires only the fault assumption of the trusted operation period, but durability requires both fault assumptions. In practice, we expect most of periods are healthy but there are occasional unhealthy periods, thus most of writes are accepted during healthy periods.

There are additional assumptions we make: 1) there is at least one correct replica during any two consecutive regular operation periods across the in-between trusted operation period, and 2) the entire immutable state of a replica can be checked locally and can be fetched from another replica if necessary during a regular operation period.

3.4.3 Two-phase Approach

The tiered Byzantine fault model can be justified by our approach that uses proactive recovery and a small trusted primitive that allows modifications only in trusted operations phases. We take a two-phase approach where service and proactive recovery phases alternate (Figure 3.10). A service phase operates under the regular operation fault bound, but a proactive recovery phase operates under the trusted operation fault bound.

We use a small trusted primitive that stores a tiny piece of states for the root of trust. This primitive is simple, easy to implement, and easy to formally verify; thus, it is fault-free. However, applications using this primitive can be faulty, and may install incorrect states. The states are used both in service and proactive recovery phases, but the service phase only reads the states and does not modify them, thus it cannot introduce faults into the states stored in the primitive even though the application that uses it is faulty. Moreover, before modifying the states, the proactive recovery phase reboots and reloads from a clean, read-only medium in order to make sure to avoid accumulated faults. Therefore, the states of the primitive can have the tighter trusted operation fault bound.

This approach implies an operation model different from traditional models. For fault tolerance, we trade off availability by using proactive recovery, and trade off interactivity of writes that involve modifications to the states of the primitive by allowing such modifications only during proactive recovery phases.

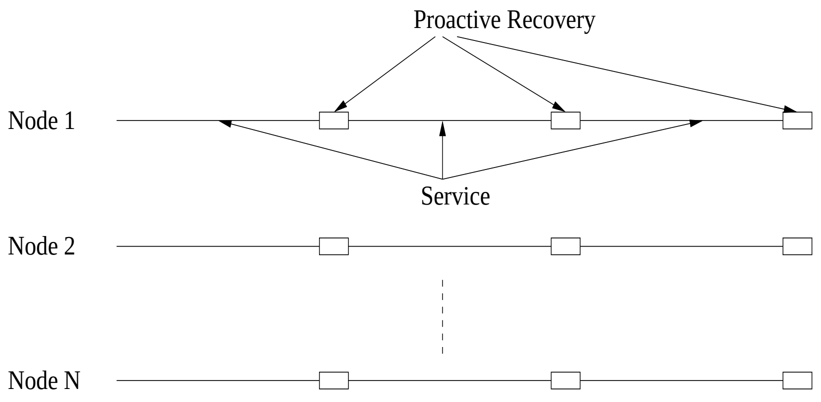

Figure 3.10: Operation of TM. Each replica alternates between a service phase and a proactive recovery phase.

3.5 TimeMachine

TimeMachine (TM) is our proposed solution to long-term authenticity in the tiered Byzantine-fault model. We present a system with a minimal `Add`/`Get` interface to focus on achieving HWICR under the tiered Byzantine fault model. Clients can invoke TM's `Add`$(name, value)$ interface method to store and preserve a particular name-value pair, if no such name is already being preserved. The `Get`$(name)$ method obtains any stored name-value pair by that name, or indicates none exists. In the context of digital preservation, a user may `Add` the name of a persistent Uniform Resource Locator (URL) as the name and the hash corresponding to the content at the URL as the value. In the future, another user can look up the content hash for the particular URL and use it to verify any result obtained via a durable bitstore such as OceanStore or Glacier. Note that there is no way to remove a name-value pair from the system.

TM is a replicated service running on replicas $R = \{1, \ldots, N\}$. Clients communicate with TM through a public network. Each replica alternates between two modes of operation (Figure 3.10): during most of the time, it operates in its service phase, serving `Get` requests from clients, buffering `Add` requests from clients, and continuously running a storage audit and repair background process; periodically (say every hour or every day), the replica reboots securely into its proactive recovery phase, during which it is only reachable by other replicas in the system, and serves buffered `Add` requests. TM is intended for a well provisioned, low-churn node infrastructure. Since membership churn is low, the membership of nodes can be managed manually.

3.5.1 Components

TM depends on the following building blocks:

Cryptography: We assume standard cryptographic primitives for symmetric and asymmetric signing, and for hashing. We also assume that in the short-term (say on the order of a calendar year), breaking such primitives through brute-force attacks is intractable for the good and the bad guys alike. Therefore, the adversary cannot produce collisions for hash functions or forge previously unseen signatures for private signing keys he does not possess. For brevity, we do not discuss more efficient authentication primitives such as MACs. $\langle M \rangle_i$ denotes a message M signed by principal i. Replica i's trusted hardware device is principal i'.

PBFT: The Practical BFT protocol [CL02] allows the implementation of replicated state machines under bounded Byzantine faults. Informally, it offers a synchronous `Invoke`(*request*) method, that returns a *response*. A client uses this method to submit application requests to a replicated state machine, and eventually receives replies containing the result of its request. The protocol guarantees that, as long as no more than a third of all replicas in the system are faulty, clients will receive replies to their requests that are equivalent to interacting with a single, correct, sequential server atomically executing requests on the application state (*linearizability*).

Furthermore, the protocol offers replicas an `Execute`(*request*) callback, allowing the application code that processes client requests to be executed. The system guarantees that the same client requests will be passed to the `Execute` method of every non-faulty replica in the same order.

Trusted Hardware: TM relies on the existence of a trusted hardware device on every replica in the system. This device is an extension of a standard TPM module, which enables trustworthy attestations about the system state contained within the module. We describe below what module-local state is required by TM.

To help conduct periodic recovery operations, the hardware device contains a time source (this can be a regular, monotonic, crystal-based clock source with an upper bound on drift, or an external trusted time source received by the device). A hardware watchdog, also contained within, uses this time source to trigger proactive recovery periodically, by causing the host to reboot from read-only media. This hardware watchdog has a *mode* bit associated with it. This bit is used to indicate that the system is in proactive recovery mode, and cannot be set in any fashion other than by triggering the watchdog. The mode bit can, however, be reset by the operating system. This is sometimes called a *sticky register*.

Finally, we include in the trusted hardware device an extension for *Secure Append-Infrequently Memory* (SAIM). A SAIM implements a sequenced queue: a queue of a small number (e.g., $10-20$) of fixed-length (e.g., 32-64 bytes), persistent entry slots. The write interface to a SAIM is that of a FIFO queue: an $\texttt{Append}(v)$ adds value v to the front of the queue, causing the oldest value to be dropped off the back of the queue. However, appends to a SAIM are *rate-limited*: if the last append occurred at time t_1, no subsequent append can succeed before time $t_2 > t_1 + \mathcal{T}$, where $\mathcal{T}$ is a fixed configuration parameter. The read interface of SAIM is not rate-limited, and allows the attested (fresh) retrieval of any slot (indexed by position in the queue); a $\texttt{Lookup}(l,z)$, where l is the position in the queue requested and z is a nonce used for freshness (typically provided by clients), returns $\langle l, v_l, z, t, m \rangle_{i'}$, where v_l is the value currently occupying the l-th slot of the SAIM, t is the internal time in the device, m is the current mode bit, and i' is the hardware device principal.

Authenticated Search Tree (AST): An AST [BLL00] is an incremental mechanism for maintaining cryptographic digests over sorted data sets (such as name-value pairs sorted by name). An AST extends the traditional Merkle tree concept for search trees. Every node contains a name-value pair and an authentication label. The label for an AST node is computed by hashing together its content and the labels of all child nodes. A correct binary AST guarantees that data items from the collection appear in the left subtree of a node if and only if they precede its own content in the search order; similarly for the right subtree. The label of the tree root is a cryptographic digest for the entire contents of the tree: it is *collision-resistant*, which means it is intractable to find two different data sets yielding the same AST digest and, as a result, it can serve as a *commitment* on the contents of the AST.

As with Merkle trees, a succinct *proof* can be generated showing that a particular name-value pair appears within an AST with a root label. Unlike Merkle trees, an AST can

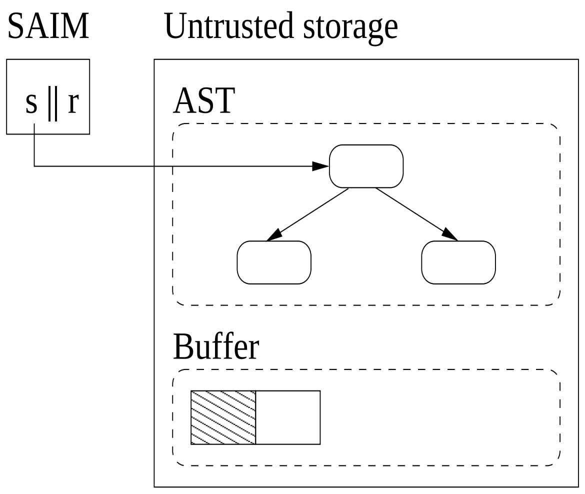

Figure 3.11: A TM node contains a SAIM, a buffer to hold Add requests temporarily, and an AST that maintains committed bindings. The arrows indicate self-verifying pointers, which mean a hash value of a starting point is the hash of a pointed block. SAIM stores the AST root digest and a sequence number.

also prove succinctly that a name-value pair *does not appear* within it, by showing the contiguous appearance of the two keys immediately preceding and following that name in the sort order. Proofs have logarithmic length in the size of the name-value pairs contained within a tree.

3.5.2 TM Design

Each TM replica maintains an AST in regular (untrusted) storage containing its collection of name-value pairs sorted by name, a buffer of received but as yet uncommitted client requests for adding new name-value pairs also in untrusted storage, and a single-slot SAIM within its trusted hardware device storing values of the form $\langle s, r \rangle$, where s is the latest AST digest and r is an integer sequence number (Figure 3.11). Since we only use a single-slot SAIM, all attestations contain $l = 0$ as the queue position. Though in this instantiation of TM we use a single-slot SAIM for simplicity, we sketch extensions that take advantage of multiple slots later. Finally, replicas know each other's public keys and hardware device public keys.

In the trusted hardware device, a watchdog timer is set to $\mathcal{D}$ and the SAIM inter-append delay is set to $\mathcal{T} > \mathcal{D}$, to ensure that a new value cannot be appended into SAIM until after the watchdog's next expiration, bringing the replica into its next proactive recovery phase.

Service Phase

When a client c invokes `Add`(n,v) to insert a binding between name n and value v, the TM client proxy code multicasts $\langle \textsc{Add}, n, v, z, c\rangle_c$ to R, where z is a random nonce used for freshness. The client waits asynchronously for replies in $\langle \textsc{ReplyAdd}, i, p_i, \langle 0, \langle s_i, r_i\rangle, z, t, m\rangle_{i'}\rangle$ messages containing a 0-th slot SAIM `Lookup` attestation, where i is a replica identifier and p_i is an AST membership proof as described above. A reply is valid if p_i verifies the existence of the name-value pair (n,v) within an AST with digest s_i, and the attestation is correctly signed by the sender's SAIM. As soon as a client proxy obtains $f+1$ valid matching replies from distinct replicas, all confirming the addition of the same name-value pair, it accepts the request as *complete* and notifies the application.

Handling of $\textsc{Get}$ requests by client proxies is similar. The proxy multicasts $\langle \textsc{Get}, n, z, c\rangle_c$ messages to R and waits for $f+1$ $\langle \textsc{ReplyGet}, i, v, p_i, \langle 0, \langle s_i, r_i\rangle, z, t, m\rangle_{i'}\rangle$ valid messages confirming that (n,v) is within the AST described in each reply, or that (n,v) does not exist in the AST described in each reply.

A replica handles a $\textsc{Get}$ by looking it up by name in its local AST and producing an existence/non-existence proof, accompanied by its latest SAIM, signed by the trusted hardware device. During the service phase, a replica only buffers $\textsc{Add}$s, which it handles during the proactive recovery phase. Note, however, that the replica replies to $\textsc{Add}$s for already assigned names immediately, returning the existing mapping. During the next service phase, the replica responds to newly inserted $\textsc{Add}$s with a $\textsc{ReplyAdd}$ message. Replies to $\textsc{Add}$s for previously unassigned names take time on the order of the length of the service phase.

Background Audit and Repair: In addition to a service process, TM replicas run a continuous audit and repair process in the background, ensuring that all reachable AST nodes from the AST root are correctly stored. This process is recursive, starting with the root, and performing an in-order traversal of the tree, during which a tree node is fetched from storage if still available and verified against the hash contained in the label of its parent node.

For all missing AST nodes (identified by the name of the stored name-value pair), a replica multicasts a $\langle \text{REQASTNODE}, i, n, r\rangle_i$ request to R, where i is the replica identifier and r is the latest known SAIM sequence number, waiting for at least one $\langle \text{RESPASTNODE}, n, \mathit{ASTNode}\rangle$ response. Note that the response need not be signed, since the replica can ensure its validity thanks to the recursive hashes of the AST.

Proactive Recovery Phase

When the trusted watchdog timer expires, the system begins a reboot from a read-only medium of its proactive recovery software. The main responsibility of the proactive recovery phase is to commit a new set of additions into the main service state. At the end of the proactive recovery phase, the system ensures that at least $2f+1$ replicas store the latest AST digest in SAIM.

Are We All Proactively Recovering?: All messages exchanged between replicas contain a fresh attestation fetched from the SAIM after the current proactive recovery phase began: the mode bit shown must be on, and the timestamp must be recent. Messages unaccompanied by this attestation are invalid and dropped. This is to ensure that proactive recovery operations, including invocations of PBFT, leader elections, etc., are performed by nodes who have rebooted into their proactive recovery phase. Therefore, any faults caused by such nodes are due to proactive recovery faults – which our tiered fault model assumes bounded by $\lfloor \frac{N-1}{3} \rfloor$ – rather than service phase faults, which are bounded by $N-1$ in our fault model.

Commit: At a high level, each replica packages up its pending ADDS (denoted by A) and the latest stable checkpoint (denoted by C_s) it knows ($2f+1$ matching $\langle s_i, r_i\rangle$ pairs) into a $\langle \text{BATCH}, C_s, A\rangle_i$ message, filtering out those ADDS for already assigned names, and multicasts the BATCH to R. Once a *leader* replica (defined below) collects $2f+1$ such messages including its own, it packages them into a PROPOSE message, which it submits to PBFT's `Invoke` for linearization. During the `Execute` callback of PBFT, a replica ensures the PROPOSEd set contains at least $2f+1$ batches from distinct replicas. If so, it picks the latest stable checkpoint: an AST digest in $2f+1$ matching SAIM attestations, and all ADDS contained in at least one batch (authenticated by clients). It orders the ADDS according to a consistent order (e.g., by $h(c\|n\|v)$), and processes each in that order: each name-value pair is inserted into the latest stable AST, unless a mapping for the same name already exists. The replica computes the new AST digest s_i^* for sequence number r_i^* ($> r_i$), appends it into its SAIM, and multicasts to R a $\langle \text{PRCHECKPOINT}, \langle 0, \langle s_i^*, r_i^* \rangle, 0, t, m \rangle_{i'} \rangle$ message. When a replica receives $2f+1$ matching PRCHECKPOINT messages, it stores them as a new stable checkpoint certificate. If a replica's old AST is not the latest one, it will have to perform state transfer, as described below.

When the replica obtains a new stable checkpoint certificate, it resets its watchdog timer to $\mathcal{D}$, and exits into its service phase by opening up communication with nodes other than replicas and resetting the phase-switch variable. In the beginning of the new service phase, the `Adds` remaining in the buffer are handled as described in the service phase, via the transmission of a REPLYADD message.

Leader Election: During each proactive recovery phase, the leader described above is the replica $i \equiv r \mod N$, where r is the current sequence number. A leader may misbehave, either by delaying the transmission of a PROPOSE message, or by transmitting an incorrect such message. The latter case can be detected during the `Execute` PBFT callback, as described above. A non-faulty replica can detect the former case by setting a timer as soon as it multicasts its BATCH message, which it uneventfully stops when it encounters its own BATCH as one of the batches included in a proposal during the `Execute` callback; if the timer expires, then the replica also initiates a leader change.

A leader change is similar to a batch commitment as above: every replica that wishes to initiate it multicasts a LEADERCHANGE message, which the *next* leader $i \equiv r+1 \mod N$ listens for. When the next leader has collected $2f+1$ such requests, it packages them into a single LEADERCHANGEREQUEST which it submits to PBFT; execution of this request after linearization increments the sequence number r and makes the next leader the new current leader. We omit the straightforward details on cascaded leader changes.

State Transfer: Before the phase can end, the SAIM of a replica must contain the latest stable checkpoint. A slow replica may be unable to obtain that by executing the PROPOSEd additions. However, the stable checkpoint broadcast by those replicas that were up to date allows a slow replica to append that AST digest into its SAIM, thereby catching up with others. Up-to-date replicas missing actual AST nodes can apply the repair process (described above for the service phase) to obtain only those AST nodes required for them to execute a new proposal. Those are only the AST nodes on the path from the tree root to the to-be-added tree leaves; other missing AST nodes can be repaired by the background audit and repair process during the service phase.

3.5.3 Optimizations

Tentative ADD response: When a replica receives an ADD request, it buffers its request to process during an upcoming proactive recovery phase. If the request is dropped in the network and does not reach enough replicas, the client would not receive responses it expects in the next service phase. To help the client decide its request reaches at least $2f+1$ replicas, each replica that receives an ADD request sends the client a tentative ADD acknowledgement to indicate it received the client's request.

Read for verification: A client that issues an ADD request during a service phase can issue a GET request for the name in the next service phase to check that its write is correctly done. If GET does not return $f+1$ replies that match its name-value pair from distinct replicas, the client presumes that its ADD request failed and multicasts its ADD request to R again.

3.5.4 Correctness

Theorem 7. *TM provides the HWICR property under the tiered Byzantine-fault model.*

Proof. (Sketch) In the proof, we denote by $s(r)$ the service phase of round r and by $p(r)$ the proactive recovery phase after $s(r)$. We say $s(r)$ is healthy if the number of faulty replicas is no more than f out of $3f+1$ total replicas at $s(r)$. From our assumption, proactive recovery phases are always healthy.

Without loss of generality consider a binding (n,v). We prove that if n is not in the AST and the binding is added at healthy $s(r)$, the binding is correctly read (or temporarily unavailable) at all $s(r')(r' > r)$.

At healthy $s(r)$, we say an $\texttt{Add}(n,v)$ request is accepted if there are at least $2f+1$

replicas that receive the request; clients can ensure that the request is accepted by checking authenticated tentative ADD responses. Let Q_t denote this set of replicas. At the start of $p(r)$, each replica multicasts a BATCH message to other replicas. The leader collects $2f+1$ distinct BATCH messages that form a BATCH certificate. Let Q_b denote the set of replicas that form this certificate. $Q_t \bigcap Q_b$ includes at least one non-faulty replica that receives the ADD request. Therefore, the accepted request is contained in the BATCH certificate.

In addition, we show that at $p(r)$ the BATCH certificate contains the stable checkpoint ($2f+1$ matching SAIM attestations) of $p(r-1)$. At $p(r-1)$, there are at least $2f+1$ replicas that agree on the BATCH certificate of $p(r-1)$ via PBFT. Let Q_p denote this set of replicas. $Q_p \bigcap Q_b$ includes at least one non-faulty replica that includes the stable checkpoint SAIM attestation. Therefore, the BATCH certificate of $p(r)$ contains the correct stable checkpoint of $p(r-1)$.

Then, PBFT ensures that at least $2f+1$ replicas agree on the PROPOSE with the above BATCH certificate. Each such replica checks that n does not exist; if necessary, the replica can perform state transfer for this validation. If not, the replica inserts (n,v) into the AST, computes a new AST digest, and appends it to SAIM. At this point, there are at least $f+1$ correct replicas, each of which correctly adds the binding to the AST and updates its SAIM.

Now, suppose a client gets a reply certificate ($f+1$ matching SAIM attestations) of `Get`(n) at $s(r+1)$. The reply certificate contains at least one up-to-date non-faulty replica since a non-faulty replica enters $s(r+1)$ only after collecting a PRCHECKPOINT certificate. Therefore, a client correctly reads value v when it queries with n at $s(r+1)$.

Once (n,v) is inserted into TM at $p(r)$, it is clear that $p(r+1)$ carries (n,v) from $p(r)$ correctly with the same argument we make for $p(r-1)$ and $p(r)$. We can inductively argue the same holds for $p(r+i)$ and $p(r+i+1)$ for all $i \geq 0$. Therefore, when a client gets a reply certificate for `Get`(n) at all $s(r+i)$ ($i>0$), the client receives correct (n,v).

$\square$

3.6 Discussion

3.6.1 Tradeoff between Safety and Availability

There is a tradeoff between safety and availability, which is affected by changing the frequency of proactive recovery. Frequent proactive recovery improves safety since it reduces the probability of N replica faults in a service phase, but it reduces availability since the TM service is not available to clients during proactive recovery.

We define the availability of the service as

$$\text{Availability} = \frac{\text{Service time}}{\text{Service time + Proactive recovery time}}.$$

The service time is an interval between two consecutive proactive recoveries. The proactive recovery time is the sum of the reboot time and the commit time. The commit time includes the time to send BATCH messages, the agreement time, and the time to incorporate new ADD requests, which requires to check a part of the AST and to repair it if necessary. The proactive recovery time depends on the system's workload, i.e., how many new ADD requests the system process. When the service time becomes shorter, it is likely that the proactive recovery time decreases since the system receives less number of ADD requests, thus the proactive recovery time decreases. However, the reboot and agreement time does not change.

TM cannot increase the frequency of proactive recovery arbitrarily due to its background process. The interval between two consecutive proactive recoveries must be greater than the time to audit and repair the main data structure under our fault assumption. As the collection of bindings grows, this time increases. We discuss below how to partition name space to bound the collection size.

When N is large, we can increase the frequency of proactive recovery by allowing the audit and repair time to be longer than an interval between two consecutive proactive recoveries. However, this requires a stronger fault assumption in service phases to preserve our safety guarantee. If the background audit and repair takes m consecutive service phases to scan the entire data structure, we need the fault assumption that there is at least a replica that is not faulty across $m+1$ consecutive service phases.

3.6.2 Extensions

Name Space Granularity: As more bindings are added to the system, it takes more time to audit and repair the AST. An approach to increasing scalability is to partition the name space and assign different replica groups to handle each partition. Name space partitioning, however, may have further uses, e.g., to support contextualized archival collections, or to allow different replicas in a preserved name service to handle only name spaces to which its operators have access. Such heterogeneous name space partitions could be handled through an extra level of indirection, via which each TM process group deals with distinct name spaces, though a single hardware device and SAIM on a single host is shared by all local TM processes.

Membership Management: To automate membership changes securely, TM can authenticate membership information with SAIM. SAIM stores a digest of a block that contains public keys of TM members. This membership information is also agreed when PBFT is run during a proactive recovery phase. Since membership changes are reflected in SAIM, TM can securely authenticate the current members of the system.

Advanced Search: To focus on HWICR guarantees in a tiered Byzantine fault model, we present a naming service with a minimal search interface. Extending the main data structure for advanced search is our future work. For example, we can include inverted list indices for keyword search.

Re-hashing Data: Due to the enhancement of cryptanalysis and computation, a hash function that was secure in the past may not be secure any more. To keep the system's security property, it should change its hash function, regenerate hashes from data with the new hash function, and reconstructs the AST of TM. This upgrade requires a secure coordinated action between the self-verifying bitstore and TM, which is beyond the scope of our work.

3.7 Evaluation

3.7.1 Implementation

To validate our design, we developed a prototype TM implementation. We implemented the service and background audit process of TM in C/C++ on Fedora Core 6. The client and server communicate with a SFS's asynchronous implementation of SUN RPC [Sri95] in the SFSlite library [sfs]. Client-server communication are authenticated by signatures. We use NTT's ESIGN with 2048-bit keys for signatures. The client uses a proxy to perform quorum operations for `Add`/`Get` call invocation. The server maintains SAIM, A2M with rate-limited appends, an AST, and a log for buffering ADDs. We use software emulation of SAIM, which also uses NTT's ESIGN with 2048-bit keys for signatures.

We store an AST and a log on a disk using Berkeley DB [ber]. We use a binary AST to minimize the size of membership proofs [YC07]. An AST is stored as a Berkeley database with a BTREE format. Each AST node is stored as a Berkeley DB record, which contains a name, a value, a hash of its left child, and a hash of its right child. The primary key of this DB is the name, and the secondary key is the hash of the entire node content. To search a value with a name in the AST and to insert a (name, value) binding to the AST while generating a membership proof, we traverse the AST using secondary keys.

3.7.2 Experiment Results

We ran our experiments with four TM replica nodes and one client node. The nodes are PCs with 1.8GHz∼3.2GHz Pentium 4 processors, 1GB RAM and 3Com 3C905C Ethernet cards. They are connected over a dual speed 10/100Mbps 3Com switch. On a 1.8GHz machine, ESIGN signature creation and verification of 20 bytes take on average $256\mu s$ and $194\mu s$, respectively.

We initially populate server ASTs with one million name-value bindings and use a simple micro-benchmark client that sends 1000 ADD or GET requests. The maximum size of a name is 128 bytes and a value is a 20-byte SHA-1 hash. For ADD, servers store bindings to their logs and return tentative acknowledgements. For GET, servers search their ASTs and return values, AST proofs, and SAIM attestations.

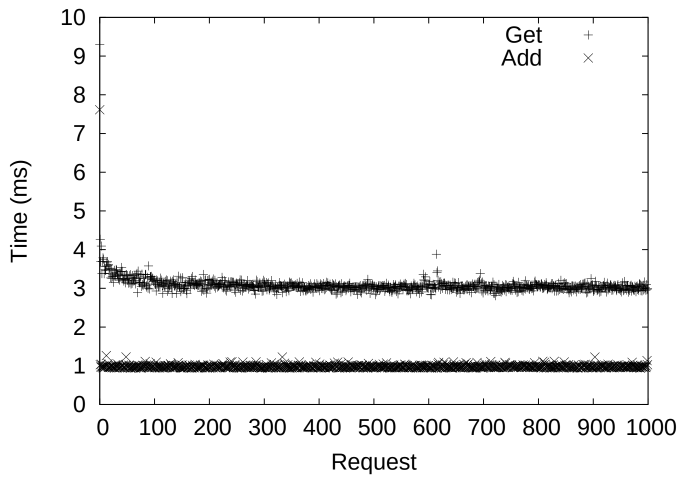

Figure 3.12: Get and Add time. In average, Get takes 3ms, and Add takes 1ms.

TM has reasonable GET and ADD latencies for long-term preservation systems. Figure 3.12 shows GET and ADD time for 1000 requests with randomly selected names. In average, GET takes 3ms, and ADD takes 1ms. GET takes more time than ADD does since it requires to access a path in the AST, which can incur to access multiple blocks in Berkeley DB.

TM availability is also high enough for long-term preservation systems. Figure 3.13 shows TM availability varying proactive recovery (PR) time and inter-PR time. When inter-PR time is 24 hours, availability is 0.9993 and 0.9931 for one-minute PR time and ten-minutes PR time, respectively. Availability decreases linearly as PR time increases. In addition, as we perform proactive recovery more frequently (i.e., inter-PR time increases), availability decreases more rapidly. For example, when PR time is ten minutes, availability drops from 0.9931 to 0.9474 as inter-PR time changes from 24 hours to 3 hours. However, when we perform proactive recovery frequently, the proactive recovery time may reduce, which mitigates frequent recovery effects, since TM needs to handle fewer ADDS. With one-minute PR time, availability becomes 0.9945 despite three-hour inter-PR time.

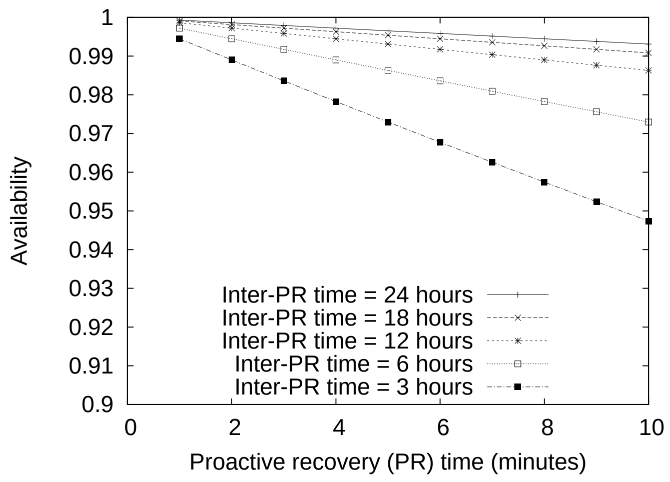

Figure 3.13: TM availability varying proactive recovery (PR) time and inter-PR time.

3.8 Future Work

There are a few enhancements we can make in proactive recovery of TM. First, we improve the fault bounds in TimeMachine, but we still have 1/3 fault bounds during proactive recovery phases. We hope to explore multiple points on the continuum of fault models through our fT-bound, in which the number of faults in T consecutive phases is bounded by fT for some fraction f, but there can be phases in which more than f replicas are faulty. Such a failure model may require multi-phase recovery and at least T SAIM slots, rather than the single-slot algorithm we described in this thesis. Second, we assume hardware clocks to invoke the proactive recovery almost at the same. Asynchronous proactive recovery that does not rely on hardware clocks might lead to more practical preservation systems.

The TM evaluation used short-running benchmarks. An evaluation of long-term usage of our systems will provide valuable insights. We hope to run TM alongside an archival service to understand better the practical applicability of this approach in a real-world archival environment.

3.9 Summary

Long-term services that operate reliably are hard to construct. This work represents a first step towards understanding better Byzantine-fault models for long-term preservation services that can be both plausible and amenable to safe solutions.

We propose a stronger property called HWICR for our preserved name service and introduce a new tiered Byzantine fault model that is suitable for long-term services. We present a naming service called TM that provides HWICR under our tiered Byzantine fault model. TM uses a unique two phase approach that alternates between regular service and trusted proactive recovery phases. By making important state changes only during proactive recovery phases, TM can tolerate more faults than previous systems do. TM puts together various components including trusted hardware to achieve stronger fault tolerance in the face of realistic threats.

Chapter 4

Selfish Replication

4.1 Overview

To tolerate faults in a single domain, systems can replicate content across multiple administrative domains. Examples are wide-area peer-to-peer file systems [DKK$^+$01, RD01, ABC$^+$02, KBC$^+$00, SKKM02], peer-to-peer caches [IRD02, GDS$^+$03], and distributed web caches [Dan98, FCAB00]. Replication of objects[1] in selected servers is widely used to enhance the performance, availability, and reliability of these systems. However, most such systems assume that servers cooperate with one another by following protocols optimized for overall system performance, regardless of the costs incurred by each server.

In reality, servers may behave selfishly — seeking to maximize their own benefit. For example, parties in different administrative domains utilize their local resources (servers) to better support clients in their own domains. They have obvious incentives to replicate objects that maximize the benefit in their domains, possibly at the expense of globally optimum behavior. It has been an open question whether these replication scenarios and protocols maintain their desirable global properties (low total social cost, for example) in the face of selfish behavior.

In this chapter, we take a game-theoretic approach to analyzing the problem of replication in networks of selfish servers through theoretical analysis and simulations. We model selfish replication as a non-cooperative game. In the *basic model*, the servers have two pos-

[1] We use the term "object" as an abstract entity that represents files and other data objects.

sible actions for each object. If a replica of a requested object is located at a nearby node, the server may be better off accessing the remote replica. On the other hand, if all replicas are located too far away, the server is better off replicating the object itself. Decisions about replicating the replicas locally are arrived at locally, taking into account only local costs. We also define a more elaborate *payment model*, in which each server bids for having an object replicated at another site. Each site now has the option of replicating an object and collecting the related bids. Once all servers have chosen a strategy, each game specifies a *configuration*, that is, the set of servers that replicate the object, and the corresponding costs for all servers.

Game theory predicts that such a situation will end up in a *Nash equilibrium*, that is, a set of (possibly randomized) strategies with the property that no player can benefit by changing its strategy while the other players keep their strategies unchanged [OR94]. However, the lack of coordination inherent in selfish decision-making may incur costs well beyond what would be globally optimum. This loss of efficiency is quantified by the *price of anarchy* [KP99]. The price of anarchy is the ratio of the social (total) cost of the worst possible Nash equilibrium to the cost of the social optimum. The price of anarchy bounds the worst possible behavior of a selfish system, when left completely on its own. However, in reality there are ways whereby the system can be guided, through "seeding" or incentives, to a pre-selected Nash equilibrium. This "optimistic" version of the price of anarchy [ADTW03] is captured by the smallest ratio between a Nash equilibrium and the social optimum.

In this work we address the following questions :

- Do pure strategy Nash equilibria exist in the replication game?
- If pure strategy Nash equilibria do exist, how efficient are they (in terms of the price of anarchy, or its optimistic counterpart) under different placement costs, network topologies, and demand distributions?
- Will a mechanism like adopting payments improve the Nash equilibria?

We show that pure strategy Nash equilibria always exist in the replication game. The price of anarchy of the basic game model can be $O(n)$, where n is the number of servers;

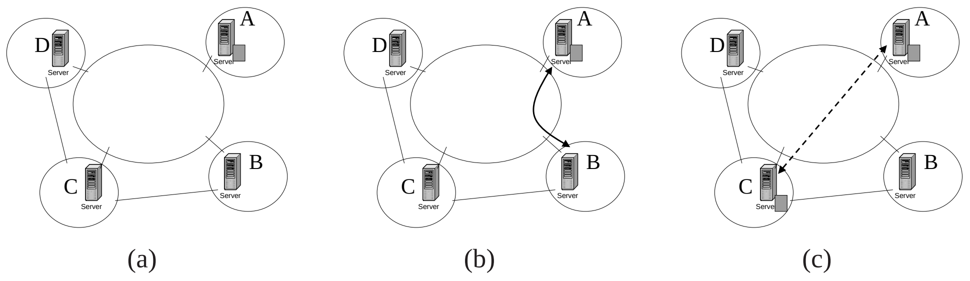

Figure 4.1: Replication. There are four servers labeled A, B, C, and D. The rectangles are object replicas. In (a), A stores an object. If B incurs less cost accessing A's replica than it would replicating the object itself, it accesses the object from A as in (b). If the distance cost is too high, the server replicates the object itself, as C does in (c). This figure is an example of our replication game model.

the intuitive reason is undersupply. Under certain topologies, the price of anarchy does have tighter bounds. For complete graphs and stars, it is $O(1)$. For D-dimensional grids, it is $O(n^{\frac{D}{D+1}})$. In the basic game, even the optimistic price of anarchy can be $O(n)$. In the payment model, however, the game can always implement a Nash equilibrium that is same as the social optimum, so the optimistic price of anarchy is one.

Our simulation results show several interesting phases. As the placement cost increases from zero, the price of anarchy increases. When the placement cost first exceeds the maximum distance between servers, the price of anarchy is at its highest due to undersupply problems. As the placement cost further increases, the price of anarchy decreases, and the effect of replica misplacement dominates the price of anarchy.

The rest is organized as follows. Section 4.2 discusses details of the basic game and analyzes the bounds of the price of anarchy. In Section 4.3 we discuss the payment game and analyze its price of anarchy. In Section 4.4 we describe our simulation methodology and study the properties of Nash equilibria observed. We discuss extensions of the game and directions for future work in Section 4.5.

4.2 Basic Game

The replication problem we study is to find a configuration that meets certain objectives (e.g., minimum total cost). Figure 4.1 shows examples of replication among four servers.

In network (a), A stores an object. Suppose B wants to access the object. If it is cheaper to access the remote replica than to replicate it, B accesses the remote replica as shown in network (b). In network (c), C wants to access the object. If C is far from A, C replicates the object instead of accessing the object from A. It is possible that in an optimal configuration it would be better to place replicas in A and B. Understanding the placement of replicas by selfish servers is the focus of our study.

The replication problem is abstracted as follows. There is a set N of n servers and a set M of m objects. The distance between servers can be represented as a distance matrix D (i.e., d_{ij} is the distance from server i to server j). D models an underlying network topology. For our analysis we assume that the distances are symmetric and the triangle inequality holds on the distances (for all servers i, j, k: $d_{ij} + d_{jk} \geq d_{ik}$). Each server has demand from clients that is represented by a demand matrix W (i.e., w_{ij} is the demand of server i for object j). When a server replicates objects, the server incurs some placement cost that is represented by a matrix α (i.e., α_{ij} is a placement cost of server i for object j).

In this study, we assume that servers have no capacity limit. As we discuss in the next section, this fact means that the replication behavior with respect to each object can be examined separately. Consequently, we can talk about *configurations* of the system with respect to a given object:

Definition 2. *A configuration X for some object O is the set of servers replicating this object.*

The goal of the basic game is to find configurations that are achieved when servers optimize their cost functions locally.

4.2.1 Game Model

We take a game-theoretic approach to analyzing the uncapacitated replication problem among networked selfish servers. We model the selfish replication problem as a non-cooperative game with n players (servers/nodes) whose strategies are sets of objects to replicate. In the game, each server chooses a pure strategy that minimizes its cost. Our focus is to investigate the resulting configuration, which is the Nash equilibrium of the game. It should be emphasized that we consider only pure strategy Nash equilibria in this work.

The cost model is an important part of the game. Let A_i be the set of feasible strategies for server i, and let $S_i \in A_i$ be the strategy chosen by server i. Given a strategy profile $S = (S_1, S_2, \ldots, S_n)$, the cost incurred by server i is defined as:

$$C_i(S) = \sum_{j \in S_i} \alpha_{ij} + \sum_{j \notin S_i} w_{ij} d_{i\ell(i,j)}. \tag{4.1}$$

where α_{ij} is the placement cost of object j, w_{ij} is the demand that server i has for object j, $\ell(i,j)$ is the closest server to i that replicates object j, and d_{ik} is the distance between i and k. When no server replicates the object, we define distance cost $d_{i\ell(i,j)}$ to be d_M—large enough that at least one server will choose to replicate the object.

The placement cost can be further divided into first-time installation cost and maintenance cost:

$$\alpha_{ij} = k_{1i} + k_{2i} \frac{UpdateSize_j}{ObjectSize_j} \frac{1}{T} P_j \sum_k w_{kj}, \tag{4.2}$$

where k_{1i} is the installation cost, k_{2i} is the relative weight between the maintenance cost and the installation cost, P_j is the ratio of the number of writes over the number of reads and writes, $UpdateSize_j$ is the size of an update, $ObjectSize_j$ is the size of the object, and T is the update period. We see tradeoffs between different parameters in this equation. For example, placing replicas becomes more expensive as $UpdateSize_j$ increases, P_j increases, or T decreases. However, note that by varying α_{ij} itself we can capture the full range of behaviors in the game. For our analysis, we use only α_{ij}.

Since there is no capacity limit on servers, we can look at each single object as a separate game and combine the pure strategy equilibria of these games to obtain a pure strategy equilibrium of the multi-object game. Fabrikant, Papadimitriou, and Talwar discuss this existence argument: if two games are known to have pure equilibria, and their cost functions are cross-monotonic, then their union is also guaranteed to have pure Nash equilibria, by a continuity argument [FPT04]. A Nash equilibrium for the multi-object game is the cross product of Nash equilibria for single-object games. Therefore, we can focus on the single object game in the rest of this work.

For single object selfish replication, each server i has two strategies — to replicate or not to replicate. The object under consideration is j. We define S_i to be 1 when server i

replicates j and 0 otherwise. The cost incurred by server i is

$$C_i(S) = \alpha_{ij} S_i + w_{ij} d_{i\ell(i,j)}(1 - S_i). \tag{4.3}$$

We refer to this game as the *basic game*. The extent to which $C_i(S)$ represents actual cost incurred by server i is beyond the scope of this work; we will assume that an appropriate cost function of the form of Equation 4.3 can be defined.

4.2.2 Nash Equilibrium Solutions

In principle, we can start with a random configuration and let this configuration evolve as each server alters its strategy and attempts to minimize its cost. Game theory is interested in stable solutions called *Nash equilibria*. A pure strategy Nash equilibrium is reached when no server can benefit by unilaterally changing its strategy. A Nash equilibrium[2] (S_i^*, S_{-i}^*) for the basic game specifies a configuration X such that $\forall i \in N, i \in X \Leftrightarrow S_i^* = 1$. Thus, we can consider a set $\mathcal{E}$ of all pure strategy Nash equilibrium configurations:

$$\begin{aligned} X \in \mathcal{E} \quad \Leftrightarrow \quad & \forall i \in N, \\ & \forall S_i \in A_i,\ C_i(S_i^*, S_{-i}^*) \leq C_i(S_i, S_{-i}^*) \end{aligned} \tag{4.4}$$

By this definition, no server has incentive to deviate in the configurations since it cannot reduce its cost.

For the basic game, we can easily see that:

$$\begin{aligned} X \in \mathcal{E} \quad \Leftrightarrow \quad & \forall i \in N, \quad \exists j \in X \quad s.t.\ d_{ji} \leq \alpha \\ and \quad & \forall j \in X, \quad \neg\exists k \in X \quad s.t.\ d_{kj} < \alpha \end{aligned} \tag{4.5}$$

The first condition guarantees that there is a server that places the replica within distance α of each server i. If the replica is not placed at i, then it is placed at another server within distance α of i, so i has no incentive to replicate. If the replica is placed at i, then the second condition ensures there is no incentive to drop the replica because no two servers separated

[2]The notation for strategy profile (S_i^*, S_{-i}^*) separates node $i's$ strategy (S_i^*) from the strategies of other nodes (S_{-i}^*).

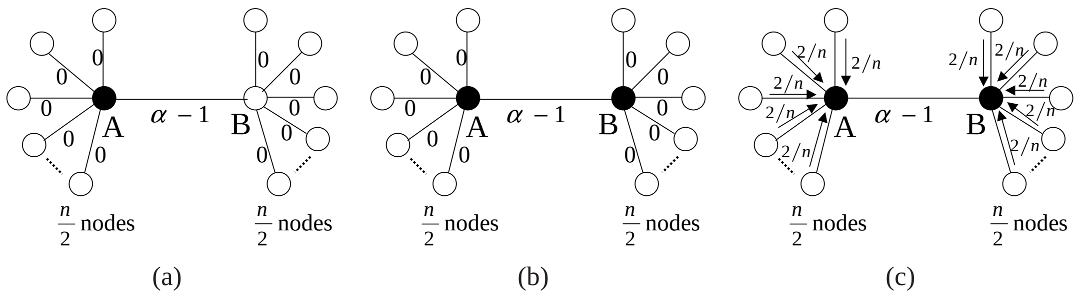

Figure 4.2: Potential inefficiency of Nash equilibria illustrated by two clusters of $\frac{n}{2}$ servers. The intra-cluster distances are all zero and the distance between clusters is $\alpha - 1$, where α is the placement cost. The dark nodes replicate the object. Network (a) shows a Nash equilibrium in the basic game, where one server in a cluster replicates the object. Network (b) shows the social optimum where two replicas, one for each cluster, are placed. The price of anarchy is $O(n)$ and even the optimistic price of anarchy is $O(n)$. This high price of anarchy comes from the undersupply of replicas due to the selfish nature of servers. Network (c) shows a Nash equilibrium in the payment game, where two replicas, one for each cluster, are placed. Each light node in each cluster pays $2/n$ to the dark node, and the dark node replicates the object. Here, the optimistic price of anarchy is one.

by distance less than α both place replicas.

4.2.3 Social Optimum

The *social cost* of a given strategy profile is defined as the total cost incurred by all servers, namely:

$$C(S) = \sum_{i=0}^{n-1} C_i(S) \tag{4.6}$$

where $C_i(S)$ is the cost incurred by server i given by Equation 4.1.

The social optimum cost, referred to as $C(S_O)$ for the remainder of the chapter, is the minimum social cost. The social optimum cost will serve as an important base case against which to measure the cost of selfish replication. We define $C(S_O)$ as:

$$C(S_O) = \min_S C(S) \tag{4.7}$$

where S varies over all possible strategy profiles. Note that in the basic game, this means varying configuration X over all possible configurations. In some sense, $C(S_O)$ represents the best possible replication behavior — if only nodes could be convinced to cooperate with one another.

The social optimum configuration is a solution of a mini-sum facility location problem, which is NP-hard [GJ79]. To find such configurations, we formulate an integer programming problem:

$$\begin{aligned}
&\text{minimize} \textstyle\sum_i \sum_j \left[\alpha_{ij} x_{ij} + \sum_k w_{ij} d_{ik} y_{ijk}\right] \\
&\quad \text{subject to} \\
&\quad\quad \forall i,j \quad \textstyle\sum_k y_{ijk} = I(w_{ij}) \\
&\quad\quad \forall i,j,k \quad x_{ij} - y_{kji} \geq 0 \\
&\quad\quad \forall i,j \quad x_{ij} \in \{0,1\} \\
&\quad\quad \forall i,j,k \quad y_{ijk} \in \{0,1\}
\end{aligned} \tag{4.8}$$

Here, x_{ij} is 1 if server i replicates object j and 0 otherwise; y_{ijk} is 1 if server i accesses object j from server k and 0 otherwise; $I(w)$ returns 1 if w is nonzero and 0 otherwise. The first constraint specifies that if server i has demand for object j, then it must access j from exactly one server. The second constraint ensures that server i replicates object j if any other server accesses j from i.

4.2.4 Analysis

To analyze the basic game, we first give a proof of the existence of pure strategy Nash equilibria. We discuss the price of anarchy in general and then on specific underlying topologies. In this analysis we use simply α in place of α_{ij}, since we deal with a single object and we assume placement cost is the same for all servers. In addition, when we compute the price of anarchy, we assume that all nodes have the same demand (i.e., $\forall i \in N\ w_{ij} = 1$).

Theorem 8. *Pure strategy Nash equilibria exist in the basic game.*

Proof. We show a constructive proof. First, initialize the set V to N. Then, remove all nodes with zero demand from V. Each node x defines β_x, where $\beta_x = \frac{\alpha}{w_{xj}}$. Furthermore, let $Z(y) = \{z : d_{zy} \leq \beta_z, z \in V\}$; $Z(y)$ represents all nodes z for which y lies within β_z from z.

Pick a node $y \in V$ such that $\beta_y \leq \beta_x$ for all $x \in V$. Place a replica at y and then remove

y and all $z \in Z(y)$ from V. No such z can have incentive to replicate the object because it can access y's replica at lower (or equal) cost. Iterate this process of placing replicas until V is empty. Because at each iteration y is the remaining node with minimum β, no replica will be placed within distance β_y of any such y by this process. The resulting configuration is a pure-strategy Nash equilibrium of the basic game. $\square$

The Price of Anarchy (POA)

To quantify the cost of lack of coordination, we use the price of anarchy [KP99] and the optimistic price of anarchy [ADTW03]. The price of anarchy is the ratio of the social costs of the worst-case Nash equilibrium and the social optimum, and the optimistic price of anarchy is the ratio of the social costs of the best-case Nash equilibrium and the social optimum.

We show general bounds on the price of anarchy. Throughout our discussion, we use $C(S_W)$ to represent the cost of worst case Nash equilibrium, $C(S_O)$ to represent the cost of social optimum, and PoA to represent the price of anarchy, which is $\frac{C(S_W)}{C(S_O)}$.

The worst case Nash equilibrium maximizes the total cost under the constraint that the configuration meets the Nash condition. Formally, we can define $C(S_W)$ as follows.

$$C(S_W) = \max_{X \in \mathcal{E}} (\alpha|X| + \sum_i \min_{j \in X} d_{ij}) \qquad (4.9)$$

where $\min_{j \in X} d_{ij}$ is the distance to the closest replica (including i itself) from node i and X varies through Nash equilibrium configurations.

Bounds on the Price of Anarchy

We show bounds of the price of anarchy varying α. Let $d_{min} = \min_{(i,j) \in N \times N, i \neq j} d_{ij}$ and $d_{max} = \max_{(i,j) \in N \times N} d_{ij}$. We see that if $\alpha \leq d_{min}$, $PoA = 1$ trivially, since every server replicates the object for both Nash equilibrium and social optimum. When $\alpha > d_{max}$, there is a transition in Nash equilibria: since the placement cost is greater than any distance cost, only one server replicates the object and other servers access it remotely. However, the social optimum may still place multiple replicas. Since $\alpha \leq C(S_O) \leq \alpha + \min_{j \in N} \sum_i d_{ij}$

Topology	***PoA***
Complete graph	1
Star	≤ 2
Line	$O(\sqrt{n})$
D-dimensional grid	$O(n^{\frac{D}{D+1}})$

Figure 4.3: PoA in the basic game for specific topologies

when $\alpha > d_{max}$, we obtain $\frac{\alpha+\max_{j\in N}\sum_i d_{ij}}{\alpha+\min_{j\in N}\sum_i d_{ij}} \leq PoA \leq \frac{\alpha+\max_{j\in N}\sum_i d_{ij}}{\alpha}$. Note that depending on the underlying topology, even the lower bound of PoA can be $O(n)$. Finally, there is a transition when $\alpha > \max_{j\in N}\sum_i d_{ij}$. In this case, $PoA = \frac{\alpha+\max_{j\in N}\sum_i d_{ij}}{\alpha+\min_{j\in N}\sum_i d_{ij}}$ and it is upper bounded by 2.

Figure 4.2 shows an example of the inefficiency of a Nash equilibrium. In the network there are two clusters of servers whose size is $\frac{n}{2}$. The distance between two clusters is $\alpha - 1$ where α is the placement cost. Figure 4.2(a) shows a Nash equilibrium where one server in a cluster replicates the object. In this case, $C(S_W) = \alpha + (\alpha - 1)\frac{n}{2}$, since all servers in the other cluster accesses the remote replica. However, the social optimum places two replicas, one for each cluster, as shown in Figure 4.2(b). Therefore, $C(S_O) = 2\alpha$. $PoA = \frac{\alpha+(\alpha-1)\frac{n}{2}}{2\alpha}$, which is $O(n)$. This bad price of anarchy comes from an undersupply of replicas due to the selfish nature of the servers. Note that all Nash equilibria have the same cost; thus even the optimistic price of anarchy is $O(n)$.

4.2.5 Analyzing Specific Topologies

We now analyze the price of anarchy (PoA) for the basic game with specific underlying topologies and show that PoA can have better bounds. We look at complete graph, star, line, and D-dimensional grid. In all these topologies, we set the distance between two directly connected nodes to one. We describe the case where $\alpha > 1$, since $PoA = 1$ trivially when $\alpha \leq 1$. A summary of the results is shown in Table 4.3.

For a complete graph, $PoA = 1$, and for a star, $PoA \leq 2$. For a complete graph, when $\alpha > 1$, both Nash equilibria and social optima place one replica at one server, so $PoA = 1$. For star, when $1 < \alpha < 2$, the worst case Nash equilibrium places replicas at all leaf nodes. However, the social optimum places one replica at the center node. Therefore, $PoA = \frac{(n-1)\alpha+1}{\alpha+(n-1)} \leq \frac{2(n-1)+1}{1+(n-1)} \leq 2$. When $\alpha > 2$, the worst case Nash equilibrium places one replica at a leaf node and the other nodes access the remote replica, and the social optimum places one replica at the center. $PoA = \frac{\alpha+1+2(n-2)}{\alpha+(n-1)} = 1 + \frac{n}{\alpha+(n-1)} \leq 2$.

For a line, the price of anarchy is $O(\sqrt{n})$. When $1 < \alpha < n$, the worst case Nash equilibrium places replicas every 2α so that there is no overlap between areas covered by two adjacent servers that replicate the object. The social optimum places replicas at least every $\sqrt{2\alpha}$. The placement of replicas for the social optimum is as follows. Suppose there are two replicas separated by distance d. By placing an additional replica in the middle, we want to have the reduction of distance to be at least α. The distance reduction is $d/2 + 2\{((d/2-1)-1)+((d/2-2)-2)+...+((d/2-d/4)-d/4)\} \geq d^2/8$. d should be at most $2\sqrt{2\alpha}$. Therefore, the distance between replicas in the social optimum is at most $\sqrt{2\alpha}$. $C(S_W) = \alpha\frac{(n-1)}{2\alpha} + \frac{\alpha(\alpha+1)}{2}\frac{(n-1)}{2\alpha} = \Theta(\alpha n)$. $C(S_O) \geq \alpha\frac{n-1}{\sqrt{2\alpha}} + 2\frac{\sqrt{2\alpha}/2(\sqrt{2\alpha}/2+1)}{2}\frac{n-1}{\sqrt{2\alpha}}$. $C(S_O) = \Omega(\sqrt{\alpha}n)$. Therefore, $PoA = O(\sqrt{\alpha})$. When $\alpha > n-1$, the worst case Nash equilibrium places one replica at a leaf node and $C(S_W) = \alpha + \frac{(n-1)n}{2}$. However, the social optimum still places replicas every $\sqrt{2\alpha}$. If we view PoA as a continuous function of α and compute a derivative of PoA, the derivative becomes 0 when α is $\Theta(n^2)$, which means the function decreases as α increases from n. Therefore, PoA is maximum when α is n, and $PoA = \frac{\Theta(n^2)}{\Omega(\sqrt{n}n)} = O(\sqrt{n})$. When $\alpha > \frac{(n-1)n}{2}$, the social optimum also places only one replica, and PoA is trivially bounded by 2. This result holds for the ring and it can be generalized to the D-dimensional grid. As the dimension in the grid increases, the distance reduction of additional replica placement becomes $\Omega(d^{D+1})$ where d is the distance between two adjacent replicas. Therefore, $PoA = \frac{\Theta(n^2)}{\Omega(n^{\frac{1}{D+1}}n)} = O(n^{\frac{D}{D+1}})$.

4.3 Payment Game

In this section, we present an extension to the basic game with payments and analyze the price of anarchy and the optimistic price of anarchy of the game.

4.3.1 Game Model

The new game, which we refer to as the *payment game*, allows each player to offer a payment to another player to give the latter incentive to replicate the object. The cost of replication is shared among the nodes paying the server that replicates the object.

The strategy for each player i is specified by a triplet $(v_i, b_i, t_i) \in \{N, \mathbb{R}_+, \mathbb{R}_+\}$. v_i specifies the player to whom i makes a bid, $b_i \geq 0$ is the value of the bid, and $t_i \geq 0$ denotes a threshold for payments beyond which i will replicate the object. In addition, we use R_i to denote the total amount of bids received by a node i ($R_i = \sum_{j:v_j=i} b_j$).

A node i replicates the object if and only if $R_i \geq t_i$, that is, the amount of bids it receives is greater than or equal to its threshold. Let I_i denote the corresponding indicator variable, that is, I_i equals 1 if i replicates the object, and 0 otherwise. We make the rule that if a node i makes a bid to another node j and j replicates the object, then i must pay j the amount b_i. If j does not replicate the object, i does not pay j.

Given a strategy profile, the outcome of the game is the set of tuples $\{(I_i, v_i, b_i, R_i)\}$. I_i tells us whether player i replicates the object or not, b_i is the payment player i makes to player v_i, and R_i is the total amount of bids received by player i. To compute the payoffs given the outcome, we must now take into account the payments a node makes, in addition to the placement costs and access costs of the basic game.

By our rules, a server node i pays b_i to node v_i if v_i replicates the object, and receives a payment of R_i if it replicates the object itself. Its net payment is $b_i I_{v_i} - R_i I_i$. The total cost incurred by each node is the sum of its placement cost, access cost, and net payment. It is defined as

$$C_i(S) = \alpha_{ij} I_i + w_{ij} d_{i\ell(i,j)}(1 - I_i) + b_i I_{v_i} - R_i I_i. \tag{4.10}$$

The cost of social optimum for the payment game is same as that for the basic game, since the net payments made cancel out.

4.3.2 Analysis

In analyzing the payment model, we first show that a Nash equilibrium in the basic game is also a Nash equilibrium in the payment game. We then present an important positive result — in the payment game the socially optimal configuration can always be implemented by a Nash equilibrium. This means that the optimistic price of anarchy in the payment game is always one. We know from the counterexample in Figure 4.2 that this is not guaranteed in the the basic game. In this analysis we use α to represent α_{ij}.

Theorem 9. *Any configuration that is a pure strategy Nash equilibrium in the basic game is also a pure strategy Nash equilibrium in the payment game. Therefore, the price of anarchy of the payment game is at least that of the basic game.*

Proof. Consider any Nash equilibrium configuration in the basic game. For each node i replicating the object, set its threshold t_i to 0; everyone else has threshold α. Also, for all i, $b_i = 0$.

A node that replicates the object does not have incentive to change its strategy: changing the threshold does not decrease its cost, and it would have to pay at least α to access a remote replica or incentivize a nearby node to replicate. Therefore it is better off keeping its threshold and bid at 0 and replicating the object.

A node that is not replicating the object can access the object remotely at a cost less than or equal to α. Lowering its threshold does not decrease its cost, since all b_i are zero. The payment necessary for another server to place a replica is at least α.

No player has incentive to deviate, so the current configuration is a Nash equilibrium. □

In fact for some graphs, the *PoA* of the payment game can be more than that of the basic game.

Now let us look at what happens to the example shown in Figure 4.2 in the best case. Suppose node B's neighbors each decide to pay node B an amount $2/n$. B does not have an incentive to deviate, since accessing the remote replica does not decrease its cost. The same argument holds for A because of symmetry in the graph. Since no one has an incentive to deviate, the configuration is a Nash equilibrium. Its total cost is 2α, the same as in the socially optimal configuration shown in Figure 4.2(b). Next we prove that indeed the payment game always has a strategy profile that implements the socially optimal configuration as a Nash equilibrium. We first present the following observation, which is used in the proof, about thresholds in the payment game.

Observation 1. *If node i replicates the object, j is the nearest node to i among the other nodes that replicate the object, and $d_{ij} < \alpha$ in a Nash equilibrium, then i should have a threshold at least $(\alpha - d_{ij})$. Otherwise, it cannot collect enough payment to compensate for the cost of replicating the object and is better off accessing the replica at j.*

Theorem 10. *In the payment game, there is always a pure strategy Nash equilibrium that implements the social optimum configuration. The optimistic price of anarchy in the payment game is therefore always one.*

Proof. Consider the socially optimal configuration ϕ_{opt}. Let N_o be the set of nodes that replicate the object and $N_c = N - N_o$ be the rest of the nodes. Also, for each i in N_o, let Q_i denote the set of nodes that access the object from i, not including i itself. In the socially optimal configuration, $d_{ij} \leq \alpha$ for all j in Q_i.

We want to find a set of payments and thresholds that makes this configuration implementable. The idea is to look at each node i in N_o and distribute the minimum payment needed to make i replicate the object among the nodes that access the object from i. For each i in N_o, and for each j in Q_i, we define

$$\delta_j = \min\{\alpha, \min_{k \in N_o - \{i\}} d_{jk}\} - d_{ji} \tag{4.11}$$

Note that δ_j is the difference between j's cost for accessing the replica at i and j's next best option among replicating the object and accessing some replica other than i. It is clear that $\delta_j \geq 0$.

Claim 1. *For each $i \in N_o$, let ℓ be the nearest node to i in N_o. Then, $\sum_{j \in Q_i} \delta_j \geq \alpha - d_{i\ell}$.*

Proof. (of claim) Assume the contrary, that is, $\sum_{j \in Q_i} \delta_j < \alpha - d_{i\ell}$. Consider the new configuration ϕ_{new} wherein i does not replicate and each node in Q_i chooses its next best strategy (either replicating or accessing the replica at some node in $N_o - \{i\}$). In addition, we still place replicas at each node in $N_o - \{i\}$. It is easy to see that cost of ϕ_{opt} minus cost of ϕ_{new}

is at least:

$$\begin{aligned} & (\alpha + \sum_{j \in Q_i} d_{ij}) - (d_{i\ell} + \sum_{j \in Q_i} \min\{\alpha, \min_{k \in N_o - \{i\}} d_{ik}\}) \\ = \quad & \alpha - d_{i\ell} - \sum_{j \in Q_i} \delta_j > 0, \end{aligned}$$

which contradicts the optimality of ϕ_{opt}. □

We set bids as follows. For each i in N_o, $b_i = 0$ and for each j in Q_i, j bids to i (i.e., $v_j = i$) the amount:

$$b_j = \max\{0, \delta_j - \varepsilon_i/(|Q_i| + 1)\}, \quad j \in Q_i \tag{4.12}$$

where $\varepsilon_i = \sum_{j \in Q_i} \delta_j - \alpha + d_{i\ell} \geq 0$ and $|Q_i|$ is the cardinality of Q_i. For the thresholds, we have:

$$t_i = \begin{cases} \alpha & \text{if } i \in N_c; \\ \sum_{j \in Q_i} b_j & \text{if } i \in N_o. \end{cases} \tag{4.13}$$

This fully specifies the strategy profile of the nodes, and it is easy to see that the outcome is indeed the socially optimal configuration.

Next, we verify that the strategies stipulated constitute a Nash equilibrium. Having set t_i to α for i in N_c means that any node in N is at least as well off lowering its threshold and replicating as bidding α to some node in N_c to make it replicate, so we may disregard the latter as a profitable strategy. By observation 1, to ensure that each i in N_o does not deviate, we require that if ℓ is the nearest node to i in N_o, then $\sum_{j \in Q_i} b_j$ is at least $(\alpha - d_{i\ell})$. Otherwise, i will raise t_i above $\sum_{j \in Q_i} b_j$ so that it does not replicate and instead accesses the replica at ℓ. We can easily check that

$$\sum_{j \in Q_i} b_j \geq \sum_{j \in Q_i} \delta_j - \frac{|Q_i|\varepsilon_i}{|Q_i| + 1} = \alpha - d_{i\ell} + \frac{\varepsilon_i}{|Q_i| + 1} \geq \alpha - d_{i\ell}.$$

Therefore, each node $i \in N_o$ does not have incentive to change t_i since i loses its payments received or there is no change, and i does not have incentive to b_i since it replicates the object. Each node j in N_c has no incentive to change t_j since changing t_j does not reduce its cost. It also does not have incentive to reduce b_j since the node where j accesses does not replicate and j has to replicate the object or to access the next closest replica, which costs at least the same from the definition of b_j. No player has incentive to deviate, so this strategy profile is a Nash equilibrium. □ □

4.4 Simulation

We run simulations to compare Nash equilibria for the single-object replication game with the social optimum computed by solving the integer linear program described in Equation 4.8 using Mosek [mos]. We examine price of anarchy (*PoA*), optimistic price of anarchy (*OPoA*), and the average ratio of the costs of Nash equilibria and social optima (*Ratio*), and when relevant we also show the average numbers of replicas placed by the Nash equilibrium (*Replica(NE)*) and the social optimum (*Replica(SO)*). The *PoA* and *OPoA* are taken from the worst and best Nash equilibria, respectively, that we observe over the runs. Each data point in our figures is based on 1000 runs, randomly varying the initial strategy profile and player order.

In our evaluation, we study the effects of variation in four categories: placement cost, underlying topology, demand distribution, and payments. As we vary the placement cost α, we directly influence the tradeoff between replicating and not replicating. In order to get a clear picture of the dependency of *PoA* on α in a simple case, we first analyze the basic game with a 100-node line topology whose edge distance is one.

We also explore transit-stub topologies generated using the GT-ITM library [ZCB96] and power-law topologies (Router-level Barabasi-Albert model) generated using the BRITE topology generator [MLMB01]. For these topologies, we generate an underlying physical graph of 3050 physical nodes. Both topologies have similar minimum, average, and maximum physical node distances. The average distance is 0.42. We create an overlay of 100 server nodes and use the same overlay for all experiments with the given topology.

In the game, each server has a demand whose distribution is Bernoulli(p), where p is the probability of having demand for the object; the default unless otherwise specified is $p = 1.0$.

Algorithm 1 Initialization for the Basic Game

```
$L_1$ = a random subset of servers
for each node $i$ in $N$ do
    if $i \in L_1$ then
        $S_i$ = 1 ; replicate the object
    else
        $S_i$ = 0
```

Algorithm 2 Move Selection of *i* for the Basic Game

```
$Cost_1 = \alpha$
$Cost_2 = \min_{j \in X-\{i\}} d_{ij}$ ; $X$ is the current configuration
$Cost_{min} = \min\{Cost_1, Cost_2\}$
if $Cost_{now} > Cost_{min}$ then
    if $Cost_{min} == Cost_1$ then
        $S_i = 1$
    else
        $S_i = 0$
```

4.4.1 Nash Dynamics Protocols

The simulator initializes the game according to the given parameters and a random initial strategy profile and then iterates through rounds. Initially the order of player actions is chosen randomly. In each round, each server performs the Nash dynamics protocol that adjusts its strategies greedily in the chosen order. When a round passes without any server changing its strategy, the simulation ends and a Nash equilibrium is reached.

In the basic game, we pick a random initial subset of servers to replicate the object as shown in Algorithm 1. After the initialization, each player runs the move selection

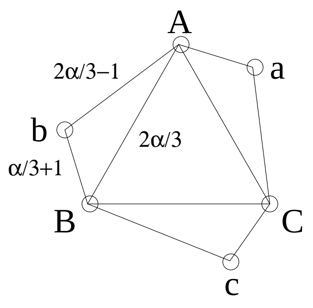

Figure 4.4: An example where the Nash dynamics protocol does not converge in the payment game.

Algorithm 3 Initialization for the Payment Game

```
$L_1$ = a random subset of servers
for each node $i$ in $N$ do
    $b_i$ = 0
    if $i \in L_1$ then
        $t_i$ = 0 ; replicate the object
    else
        $t_i$ = $\alpha$

$L_2 = \{\}$
for each node $i$ in $N$ do
    if coin toss == head then
        $M_i = \{j : d(j,i) < \min_{k \in L_1 \cup L_2} d(j,k)\}$
        if $M_i$ != $\emptyset$ then
            for each node $j \in M_i$ do
                $b_j = max\{\frac{\alpha + \sum_{k \in M_i} d(i,k)}{|M_i|} - d(i,j), 0\}$
            $L_2 = L_2 \cup \{i\}$
```

procedure described in Algorithm 2 (in algorithms 2 and 4, $Cost_{now}$ represents the current cost for node i). This procedure chooses greedily between replication and non-replication. It is not hard to see that this Nash dynamics protocol converges in two rounds.

In the payment game, we pick a random initial subset of servers to replicate the object by setting their thresholds to 0. In addition, we initialize a second random subset of servers to replicate the object with payments from other servers. The details are shown in Algorithm 3. After the initialization, each player runs the move selection procedure described in Algorithm 4. This procedure chooses greedily between replication and accessing a remote replica, with the possibilities of receiving and making payments, respectively. In the protocol, each node increases its threshold value by *incr* if it does not replicate the object. By this ramp up procedure, the cost of replicating an object is shared fairly among the nodes that access a replica from a server that does replicate. If *incr* is small, cost is shared more fairly, and the game tends to reach equilibria that encourages more servers to store replicas, though the convergence takes longer. If *incr* is large, the protocol converges quickly, but it may miss efficient equilibria. In the simulations we set *incr* to 0.1. Most of our simulation runs converged, but there were a very few cases where the simulation did not converge due to the cycles of dynamics. The protocol does not guarantee convergence within a certain

Algorithm 4 Move Selection of i for the Payment Game

```
$Cost_1 = \alpha - R_i$
$Cost_2 = \min_{j \in N-\{i\}} \{t_j - R_j + d_{ij}\}$
$Cost_{min} = \min\{Cost_1, Cost_2\}$
**if** $Cost_{now} > Cost_{min}$ **then**
    **if** $Cost_{min} == Cost_1$ **then**
        $t_i = R_i$
    **else**
        $t_i = R_i + incr$
        $v_i = argmin_j \{t_j - R_j + d_{ij}\}$
        $b_i = t_{v_i} - R_{v_i}$
```

number of rounds like the protocol for the basic game.

We provide an example graph and an initial condition such that the Nash dynamics protocol does not converge in the payment game if started from this initial condition. The graph is represented by a shortest path metric on the network shown in Figure 4.4. In the starting configuration, only A replicates the object, and a pays it an amount $\alpha/3$ to do so. The thresholds for A, B and C are $\alpha/3$ each, and the thresholds for a, b and c are $2\alpha/3$. It is not hard to verify that the Nash dynamics protocol will never converge if we start with this condition.

The Nash dynamics protocol for the payment game needs further investigation. The dynamics protocol for the payment game should avoid cycles of actions to achieve stabilization of the protocol. Finding a self-stabilizing dynamics protocol is an interesting problem. In addition, a fixed value of *incr* cannot adapt to changing environments. A small value of *incr* can lead to efficient equilibria, but it can take long time to converge. An important area for future research is looking at adaptively changing *incr*.

4.4.2 Varying Placement Cost

Figure 4.5 shows *PoA*, *OPoA*, and *Ratio*, as well as number of replicas placed, for the line topology as α varies. We observe two phases. As α increases the *PoA* rises quickly to a peak at 100. After 100, there is a gradual decline. *OPoA* and *Ratio* show behavior similar to *PoA*.

These behaviors can be explained by examining the number of replicas placed by Nash

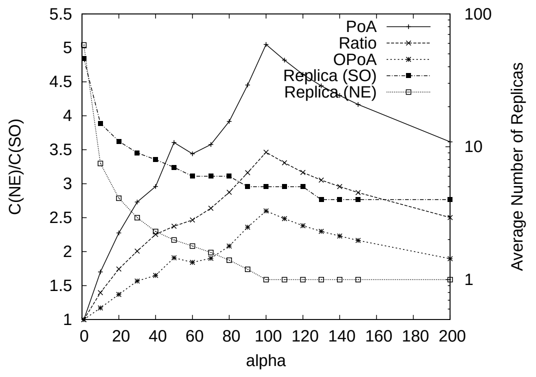

Figure 4.5: We present *PoA*, *Ratio*, and *OPoA* results for the basic game, varying α on a 100-node line topology, and we show number of replicas placed by the Nash equilibria and by the optimal solution. We see large peaks in *PoA* and *OPoA* at $\alpha = 100$, where a phase transition causes an abrupt transition in the lines.

equilibria and by optimal solutions. We see that when α is above one, Nash equilibrium solutions place fewer replicas than optimal on average. For example, when α is 100, the social optimum places four replicas, but the Nash equilibrium places only one. The peak in *PoA* at $\alpha = 100$ occurs at the point for a 100-node line where the worst-case cost of accessing a remote replica is slightly less than the cost of placing a new replica, so selfish servers will never place a second replica. The optimal solution, however, places multiple replicas to decrease the high global cost of access. As α continues to increase, the undersupply problem lessens as the optimal solution places fewer replicas.

4.4.3 Different Underlying Topologies

In Figure 4.6(a) we examine an overlay graph on the more realistic transit-stub topology. The trends for the *PoA*, *OPoA*, and *Ratio* are similar to the results for the line topology, with a peak in *PoA* at α = 0.8 due to maximal undersupply.

In Figure 4.7(a) we examine an overlay graph on the power-law topology. We observe several interesting differences between the power-law and transit-stub results. First, the *PoA* peaks at a lower level in the power-law graph, around 2.3 (at $\alpha = 0.9$) while the peak *PoA* in the transit-stub topology is almost 3.0 (at $\alpha = 0.8$). After the peak, *PoA* and *Ratio* decrease more slowly as α increases. *OPoA* is close to one for the whole range of α

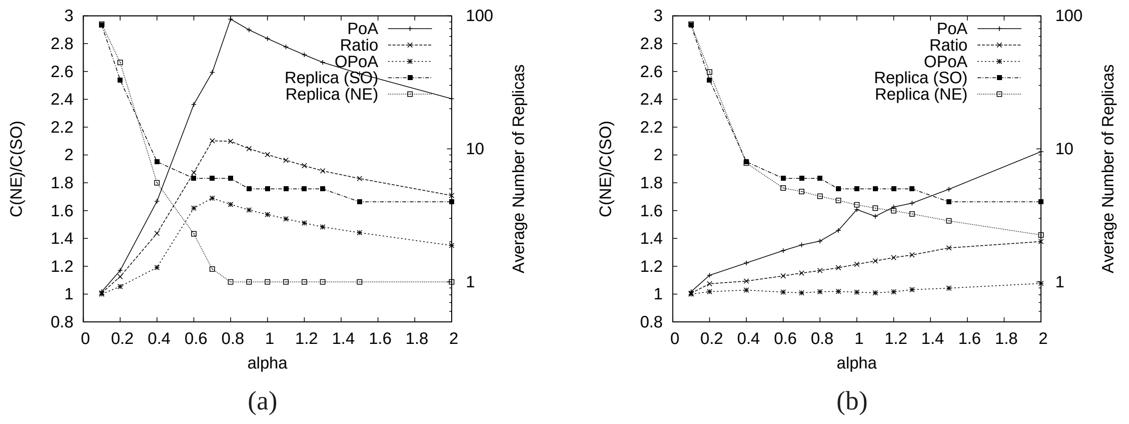

Figure 4.6: Transit-stub topology: (a) basic game, (b) payment game. We show the *PoA*, *Ratio*, *OPoA*, and the number of replicas placed while varying α between 0 and 2 with 100 servers on a 3050-physical-node transit-stub topology.

values. This can be explained by the observation in Figure 4.7(a) that there is no significant undersupply problem here like there was in the transit-stub graph. Indeed the high *PoA* is due mostly to misplacement problems when α is from 0.7 to 2.0, since there is little decrease in *PoA* when the number of replicas in social optimum changes from two to one. The *OPoA* is equal to one in the figure when the same number of replicas are placed.

4.4.4 Varying Demand Distribution

Now we examine the effects of varying the demand distribution. The set of servers with demand is random for $p < 1$, so we calculate the expected *PoA* by averaging over 5 trials (each data point is based on 5000 runs). We run simulations for demand levels of $p \in \{0.2, 0.6, 1.0\}$ as α is varied on the 100 servers on top of the transit-stub graph. We observe that as demand falls, so does expected *PoA*. As p decreases, the number of replicas placed in the social optimum decreases, but the number in Nash equilibria changes little. Furthermore, when α exceeds the overlay diameter, the number in Nash equilibria stays constant when p varies. Therefore, lower p leads to a lesser undersupply problem, agreeing with intuition. We do not present the graph due to space limitations and redundancy; the *PoA* for $p = 1.0$ is identical to *PoA* in Figure 4.6(a), and the lines for $p = 0.6$ and $p = 0.2$ are similar but lower and flatter.

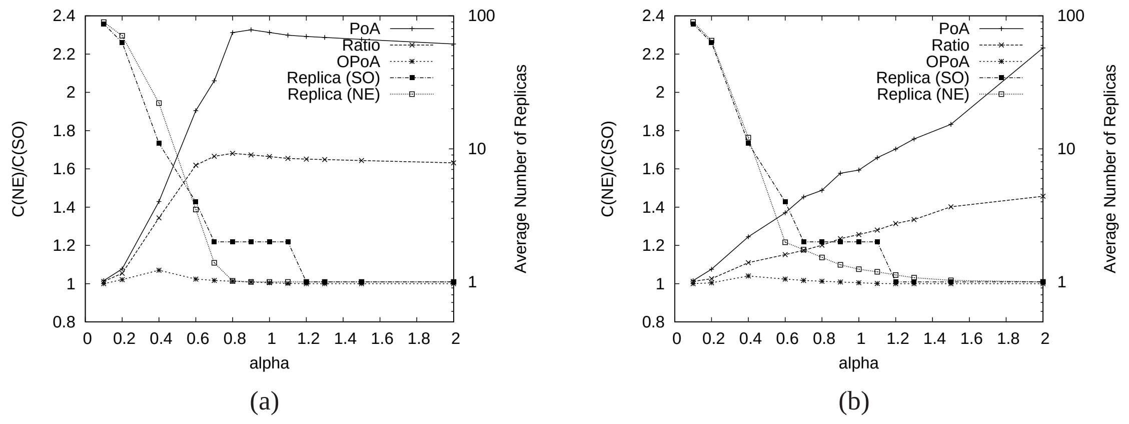

Figure 4.7: Power-law topology: (a) basic game, (b) payment game. We show the *PoA*, *Ratio*, *OPoA*, and the number of replicas placed while varying α between 0 and 2 with 100 servers on a 3050-physical-node power-law topology.

4.4.5 Effects of Payment

Finally, we discuss the effects of payments on the efficiency of Nash equilibria. The results are presented in Figure 4.6(b) and Figure 4.7(b). As shown in the analysis, the simulations achieve *OPoA* close to one (it is not exactly one because of randomness in the simulations). The *Ratio* for the payment game is much lower than the *Ratio* for the basic game, since the protocol for the payment game tends to explore good regions in the space of Nash equilibria. We observe in Figure 4.6 that for $\alpha \geq 0.4$, the average number of replicas of Nash equilibria gets closer with payments to that of the social optimum than it does without. We observe in Figure 4.7 that more replicas are placed with payments than without when α is between 0.7 and 1.3, the only range of significant undersupply in the power-law case. The results confirm that payments give servers incentive to replicate the object and this leads to better equilibria.

4.5 Discussion

We suggest several interesting extensions and directions. One extension is to consider multiple objects in the capacitated replication game, in which servers have capacity limits when placing objects. Since replicating one object affects the ability to replicate another, there is no separability of a multi-object game into multiple single object games. As studied in [GLMT04], one way to formulate this problem is to find the best response of a server by solving a knapsack problem and to compute Nash equilibria.

In our analyses, we assume that all nodes have the same demand. However, nodes could have different demand depending on objects. We intend to examine the effects of heterogeneous demands (or heterogeneous placement costs) analytically. We also want to look at the following "aggregation effect". Suppose there are $n-1$ clustered nodes with distance of $\alpha-1$ from a node hosting a replica. All nodes have demands of one. In that case, the price of anarchy is $O(n)$. However, if we aggregate $n-1$ nodes into one node with demand $n-1$, the price of anarchy becomes $O(1)$, since α should be greater than $(n-1)(\alpha-1)$ to replicate only one object. Such aggregation can reduce the inefficiency of Nash equilibria.

We intend to compute the bounds of the price of anarchy under different underlying topologies such as random graphs or growth-restricted metrics. We want to investigate whether there are certain distance constraints that guarantee $O(1)$ price of anarchy. In addition, we want to run large-scale simulations to observe the change in the price of anarchy as the network size increases.

Another extension is to consider server congestion. Suppose the distance is the network distance plus $\gamma \times (number\ of\ accesses)$ where γ is an extra delay when an additional server accesses the replica. Then, when $\alpha > \gamma$, it can be shown that *PoA* is bounded by $\frac{\alpha}{\gamma}$. As γ increases, the price of anarchy bound decreases, since the load of accesses is balanced across servers.

While exploring the replication problem, we made several observations that seem counterintuitive. First, the *PoA* in the payment game can be worse than the *PoA* in the basic game. Another observation we made was that the number of replicas in a Nash equilibrium can be more than the number of replicas in the social optimum even without payments. For example, a graph with diameter slightly more than α may have a Nash equilibrium configuration with two replicas at the two ends. However, the social optimum may place one replica at the center. We leave the investigation of more examples as an open issue.

4.6 Summary

In this chapter we introduce a novel non-cooperative game model to characterize the replication problem among selfish servers without any central coordination. We show that pure strategy Nash equilibria exist in the game and that the price of anarchy can be $O(n)$ in general, where n is the number of servers, due to undersupply problems. With specific topologies, we show that the price of anarchy can have tighter bounds. More importantly, with payments, servers are incentivized to replicate and the optimistic price of anarchy is always one. Non-cooperative replication is a more realistic model than cooperative replicating in the competitive Internet, hence this work is an important step toward viable federated replication systems.

Chapter 5

Related Work

Shared Servers: Ivy [MMGC02] is a read/write peer-to-peer file system shared by multiple users. A file system consists of a set of logs, each of which is owned by a participant who has a public-private key pair. A log is a list of immutable log records. Each log has a log-head that points to the most recent log record and the log-head is signed by the private key. A write appends a new log record and modifies the log-head to point to it. A read scans all log records owned by all participants of the file system to find appropriate information. A malicious server hosting the log-head can easily mount forking attacks by concealing log records depending on clients. With A2M, we can ensure that a malicious server tells the same sequence of log records including the most recent one. Note, however, that Ivy depends on a distributed hash table underneath, and any “strengthening” of the protocol must be predicated on a DHT with provable routing guarantees.

Plutus [KRS$^+$03] is a shared storage system that enables file sharing without placing much trust in the file servers. All data is encrypted and stored and key distribution is decentralized. A file system is represented by a hash tree, and the root hash of the tree is signed. Plutus is also vulnerable to forking attacks wherein a malicious server can show different file system states to different clients.

Replicated State Machines: Byzantine-fault tolerant state machine replication has received much attention since PBFT [CL99] added the word “practical” in its title. Researchers have proposed several improvements on PBFT such as proactive recovery (PBFT-PR [CL02]), abstraction to tolerate non-determinism [RCL01], and an architecture that separates execution from agreement to improve performance and confidentiality [YMV$^+$03]. In all cases, however, no improvement can offer liveness and safety beyond the uniform $\lfloor\frac{N-1}{3}\rfloor$ fault bound. In [YMV$^+$03], the architecture uses two groups of replicas – N agreement and M execution replicas – by dividing functionalities. This architecture can tolerate $\lfloor\frac{N-1}{3}\rfloor$ faults and $\lfloor\frac{M-1}{2}\rfloor$ faults. A2M-enabled protocols divide functionalities into committing a sequence of protocol steps to A2M and performing an original protocol. A2M-PBFT-EA can tolerate $\lfloor\frac{N-1}{2}\rfloor$ faults out of N total replicas since A2M is in a trusted computing base. Compared to agreement replicas, A2M is a small, general purpose mechanism that is applicable to various protocols to defend against equivocation.

Recently, BFT2F [LM07], a PBFT variant uses some of the ideas in SUNDR to provide linearizability and liveness up to $\lfloor\frac{N-1}{3}\rfloor$ faults, and a weaker safety property called fork* consistency without liveness for up to $2\lfloor\frac{N-1}{3}\rfloor$ faults, relying on clients' help to protect consistency. With the help of A2M, A2M-PBFT-E can instead guarantee linearizability up to $2\lfloor\frac{N-1}{3}\rfloor$ faults, and A2M-PBFT-EA guarantee both linearizability and liveness up to $\lfloor\frac{N-1}{2}\rfloor$ faults.

In loosely related work, BAR [AAC$^+$05] fault tolerance contains a notion of protocol action commitment (to a quorum maintained by replicas themselves) to capture rational behavior. Also, PeerReview [HKD07], CATS [YC07], and Timeweave [MB02] use authenticated histories to allow fault detection given a replica's self-inconsistent history; this might be a helpful mechanism to allow A2M-based protocols to recover even when the safety fault bound is (temporarily) violated.

A2M-PBFT-EA bears a close resemblance to Paxos [Lam98] in that they both require quorum size $\lfloor\frac{N-1}{2}\rfloor + 1$. Paxos assumes benign faults, and it is live as long as fewer than one half replicas are faulty but is safe with up to N faults. In contrast, A2M-PBFT-EA assumes Byzantine faults, but thanks to A2M a faulty node can stop or lie consistently to other replicas. A2M-PBFT-EA is both safe and live when fewer than one half replicas are faulty, but when this assumption is violated, there is no guarantee on safety and liveness.

Symmetric-Fault Tolerance: Researchers have described *symmetric faults* [TP88] as a specialization of Byzantine faults, and shown that for agreement protocols, a hybrid fault model that is a mixture of non-malicious faults (of size b), malicious symmetric faults (of size s), and malicious asymmetric faults (of size a) can lead to more flexible tolerance guarantees. In [TP88], a modified version of the classic synchronous Oral Messages (OM) agreement algorithm can tolerate $a+s+b$ faults when $N > 2a+2s+b+r$ (for $a \leq r$) where r is the number of rounds of message exchange excluding initial transmission. Follow-on work includes analyses of fault bounds on synchronous and asynchronous approximate agreement under the hybrid fault model [KA94, AK96]. In contrast, we focus on providing a practical, generic, small primitive that prevents equivocation to limit Byzantine hosts to behave symmetrically and constructing replicated state machine and shared storage protocols with better fault tolerance in a weak synchrony environment. We hope to explore further whether A2M can be used as a systematic way to make Byzantine faults symmetric, admitting simpler protocols with greater fault tolerance.

Abstract Shared Objects: Fleet [MR00] uses a consensus protocol by performing read and append operations on Timed Append-Only Arrays (TAOAs), which are single-writer multi-reader objects to which clients can append values and from which clients can read values. Each appended value is tagged with a logical timestamp vector. A TAOA is emulated by a distributed client-server protocol built atop a b-masking quorum system [MR97], which requires $N > 4b$ to tolerate b Byzantine faults. Unless this fault bound is violated, a TAOA provides the following properties: values are appended in a sequential order; values appended are not modified or deleted; and timestamps partially capture the order of values that different clients append. In contrast, A2M is a local primitive that can be used to enforce a node to commit to a sequential order of operations. Our goal is to slightly grow the trusted computing base to strengthen distributed trustworthy abstractions such as replicated state machines and shared storage built atop the base. In fact, implementing Fleet's TAOA and consensus protocol could be simplified if servers employ A2Ms.

Trusted Devices: Trusted hardware, such as today's commodity Trusted Platform Module (TPM) hardware developed by the Trusted Computing Group [tcg] has been previously proposed, implemented, and marketed as a way to securely boot a sensitive host with approved, bug-free software. Operations performed by the TPM are authenticated using a private signing key that resides on the module and cannot be retrieved or modified without physically destroying the module. Unfortunately, software is not bug-free, and even if correctly loaded at secure boot time, it can be overcome by exploits such as buffer overflows. As a result, while existing secure hardware can make machines strictly harder to compromise, it does not obviate the need for Byzantine-fault tolerant systems, nor does it improve their safety and liveness properties: it makes the likely number of faults smaller, but does not improve resilience against a given number of faults.

The hardware is *tamper-resistant*, which means that its cryptographic keys and its correct operation cannot be compromised remotely or physically; at worst, the host computer can be made inoperative, but its trusted hardware cannot be coerced to attest false statements. As long as the hardware manufacturer, who assigns and certifies keys used by a trusted device, does not leak its private keys, a device can secure and attest to others the software booting on a computer. Unfortunately, a hardware manufacturer who is trusted today may not be trusted 30 years from now; its private keys may have been leaked or compromised, or even the manufacturer itself may have taken to unwholesome behavior. “Trusted hardware” is a term that must be defined carefully in a long-term context.

Proactive Recovery: Proactive recovery for BFT systems [CL02] periodically reboots a potentially buggy machine with a fresh installation of the software from a read-only medium, flushing any runtime code damage that may have been done by bug exploits since the last reboot. Whereas without proactive recovery, BFT systems have a *vulnerability window* – the time extent during which the total number of faults must be bounded – that spans the entire lifetime of the system, with proactive recovery this window shrinks to a much shorter extent, typically on the order of a few inter-recovery intervals; as long as faults are spread so that no vulnerability window contains more than the bound, all is good. Nevertheless, if the bound is ever violated during any vulnerability window, guarantees are lost forever after.

Preservation: OceanStore [KBC$^+$00] and Glacier [HMD05] are distributed storage systems that use replication of self-certifying data to provide data durability. As far as we know, LOCKSS [MRG$^+$05] is the only proposal for digital preservation not requiring an inviolable N/3 bound on faults in preserving non-self-certifying data. LOCKSS, however, is probabilistic in nature and does not yet provide hard safety or liveness guarantees.

Certified Accountable Tamper-evident Storage (CATS) [YC07] is a service that provides strong accountability of actions done by the server and clients. Its approach is not to mask faults through replicated servers, but to detect faults and punish actors responsible for the faults. It uses an auditing scheme that catches server rollback attacks probabilistically and its snapshot creation frequency depends on the request rate and write sharing.

Replica Placement: The placement of replicas in the caching problem is the most important issue. There is much work on the placement of web replicas, instrumentation servers, and replicated resources. All protocols assume obedience and ignore participants' incentives. In [GHI$^+$01], Gribble et al. discuss the data placement problem in peer-to-peer systems. Ko and Rubenstein propose a self-stabilizing, distributed graph coloring algorithm for the replicated resource placement [KR03]. Chen, Katz, and Kubiatowicz propose a dynamic replica placement algorithm exploiting underlying distributed hash tables [CKK02]. Douceur and Wattenhofer describe a hill-climbing algorithm to exchange replicas for reliability in FARSITE [DW01]. RaDar is a system that replicates and migrates objects for an Internet hosting service [RRRA99]. Tang and Chanson propose a coordinated en-route web caching that caches objects along the routing path [TC02]. Centralized algorithms for the placement of objects, web proxies, mirrors, and instrumentation servers in the Internet have been studied extensively [LGI$^+$99, QPV01, JJJ$^+$00, JJK$^+$01].

The facility location problem has been widely studied as a centralized optimization problem in theoretical computer science and operations research [MF90]. Since the problem is NP-hard, approximation algorithms based on primal-dual techniques, greedy algorithms, and local search have been explored [JV99, MP00, MYZ02]. Our caching game is different from all of these in that the optimization process is performed among distributed selfish servers.

Game Theory: There is little research in non-cooperative facility location games, as far as we know. Vetta [Vet02] considers a class of problems where the social utility is submodular (submodularity means decreasing marginal utility). In the case of competitive facility location among corporations he proves that any Nash equilibrium gives an expected social utility within a factor of 2 of optimal plus an additive term that depends on the facility opening cost. Their results are not directly applicable to our problem, however, because we consider each server to be tied to a particular location, while in their model an agent is able to open facilities in multiple locations. Note that in that paper the increase of the price of anarchy comes from oversupply problems due to the fact that competing corporations can open facilities at the same location. On the other hand, the significant problems in our game are undersupply and misplacement.

In a recent paper, Goemans et al. analyze content distribution on ad-hoc wireless networks using a game-theoretic approach [GLMT04]. As in our work, they provide monetary incentives to mobile users for caching data items, and provide tight bounds on the price of anarchy and speed of convergence to (approximate) Nash equilibria. However, their results are incomparable to ours because their pay-off functions neglect network latencies between users, they consider multiple data items (markets), and each node has a limited budget to cache items.

Cost sharing in the facility location problem has been studied using *cooperative* game theory [GS00, PT03, DMV03]. Goemans and Skutella show strong connections between fair cost allocations and linear programming relaxations for facility location problems [GS00]. Pál and Tardos develop a method for cost-sharing that is approximately budget-balanced and group strategyproof and show that the method recovers 1/3 of the total cost for the facility location game [PT03]. Devanur, Mihail, and Vazirani give a strategyproof cost allocation for the facility location problem, but cannot achieve group strategyproofness [DMV03].

Chapter 6

Conclusion and Future Work

To conclude this thesis, we summarize key results of this work and describe potential future research directions.

6.1 Summary

In this thesis, we explored mechanisms to tolerate misbehavior – either Byzantine or selfish behavior – in replicated systems.

We first investigated how minimal trusted primitives can improve Byzantine fault tolerance of replicated and centralized systems in practical ways. We proposed Attested Append-Only Memory (A2M), a trusted system facility that prevents equivocation. A service using A2M will always provide the same (verifiable) answer to a given question. A2M provides the abstraction of a trusted log that keeps the immutable history (e.g., linearized executed operations). Using A2M, we improved upon the state of the art in Byzantine-fault tolerant replicated state machines, producing A2M-enabled protocols (variants of Castro and Liskov's PBFT) that remain correct (linearizable) and keep making progress (live) even when half the replicas are faulty, in contrast to the previous upper bound. We also presented an A2M-enabled single-server protocol that guarantees linearizability despite server faults. Our prototype demonstrates that this fault tolerance improvement is achieved with minor performance overhead.

Second, we addressed fault tolerance issues of long-running applications such as digital preservation systems. Due to the operating time scale, traditional homogeneous approaches to this problem are thus very likely to violate any fault bound due to short-term overwhelming faults and to lose safety.

By taking a fresh look at the traditional service properties and Byzantine fault models, we proposed a new service property called HWICR that fits well to long-term services and adapts the traditional Byzantine fault model to a tiered model that is inspired by different levels of security assurance. We showed how to split a Byzantine-fault tolerant service into a service of alternating service and proactive recovery phases.

In particular, we explored a long-term naming service that preserves mappings between human readable names and authenticators, which is a missing piece in the current archival storage literature. We presented TimeMachine, a Byzantine-fault tolerant preserved name service that uses simple, easy-to-build trusted hardware to preserve data that are not self-verifying. TimeMachine splits system operation into alternating phases of service and proactive recovery; it can survive up to N-1 faults out of N replicas during normal operation and up to $\lfloor \frac{N-1}{3} \rfloor$ faulty replicas during proactive recovery phases. The service phase serves client queries and audits self-verifying blocks. The proactive recovery phase makes important state changes by incorporating new additions received during the previous service phase.

Third, we examined replication in multiple administrative domains (MADs) that have incentives to behave rationally. We take a game-theoretic approach to characterize the impacts of rational behavior on the efficiency of replication. We show that selfish replication results in high access cost; when there is no topology restriction, the inefficiency measure (i.e., the price of anarchy) is proportional to the size of the network. However, with payment, the best achievable replication configuration is always socially optimal. Our findings suggest that a proper incentive protocol can lead to a good social behavior in MAD applications.

6.2 Future Work

We believe that A2M is a start in the research direction of using small and generic trusted primitives to achieve better system properties. We demonstrated that including A2M in a trusted computing base can benefit in building distributed trustworthy services such as replicated state machines and shared storage via improved fault tolerance. Investigating other trusted abstractions and their translation to practical system facilities is a promising future work; examples include trusted logical clocks for session guarantees and trusted version vectors for optimistic replication.

We applied A2M to systems that operate in Byzantine environments. An interesting direction is to apply A2M to systems in rational environments where an individual behaves selfishly to maximize its gain but is not Byzantine. It is an open question whether A2M's prevention of equivocation among rational agents can lead to more desirable equilibria.

We implemented A2M in a library. We hope to explore other implementation scenarios such as VMM and trusted hardware. We hope to implement a Xen A2M driver for applications running on top of Xen. In addition, we hope to explore the cost of adding A2M to a commercial TPM-like environment.

We explored a design space for long-term Byzantine fault tolerance with TM. We focused on digital preservation applications, but the HWICR property, tiered fault model, and two-phase approach may well be applicable to state machine replication of other non-interactive applications.

There are a few enhancements we can make in proactive recovery of TM. First, we improve the fault bounds in TimeMachine, but we still have 1/3 fault bounds during proactive recovery phases. We hope to explore multiple points on the continuum of fault models through our fT-bound, in which the number of faults in T consecutive phases is bounded by fT for some fraction f, but there can be phases in which more than f replicas are faulty. Such a failure model may require multi-phase recovery and at least T SAIM slots, rather than the single-slot algorithm we described in this thesis. Second, we assume hardware clocks to invoke the proactive recovery almost at the same. Asynchronous proactive recovery that does not rely on hardware clocks might lead to more practical preservation systems.

The TM evaluation used short-running benchmarks. An evaluation of long-term usage of our systems will provide valuable insights. We hope to run TM alongside an archival service to understand better the practical applicability of this approach in a real-world archival environment.

Finally, we addressed problems of replication among rational nodes. Our results are mostly existence proofs; thus, developing a practical payment protocol is future work. Our model considers only rational nodes, but in real world, we would have both selfish and Byzantine nodes. Considering both Byzantine and rational nodes has gotten attention in systems research community (e.g., BAR [AAC$^+$05]). Tackling this problem with *practical* solutions would be a long-term challenge.

Bibliography

[AAC+05] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proc. of SOSP*, 2005.

[ABC+02] 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 *Proc. of OSDI*, 2002.

[AD04] Todd W. Arnold and Leendert P. Van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. 48(3/4):475–487, 2004.

[ADTW03] Elliot Anshelevich, Anirban Dasgupta, Éva Tardos, and Tom Wexler. Near-optimal network design with selfish agents. In *Proc. of STOC*, 2003.

[AEMGG+05] Michael Abd-El-Malek, Greg Ganger, Garth Goodson, Michael Reiter, and Jay Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. of SOSP*, 2005.

[Age] National Security Agency. Global information grid (gig). `http://www.nsa.gov/ia/industry/gig.cfm`.

[AK96] Mohammad H. Azmanesh and Roger M. Kieckhafer. New hybrid fault models for asynchronous approximate agreement. *IEEE Trans. on Computers*, 45(4):439–449, 1996.

[amt] Intel Active Management Technology (AMT).

[BBC$^+$04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. of NSDI*, March 2004.

[BCG$^+$06] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the trusted platform module. In *Proc. of USENIX Security*, 2006.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of SOSP*, 2003.

[ber] Berkeley DB. `http://www.oracle.com/database/berkeley-db/`.

[BLL00] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In *Proc. of CCS*, 2000.

[BSR$^+$06] Mary Baker, Mehul Shah, David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, TJ Giuli, and Prashanth Bungale. A fresh look at the reliability of long-term digital storage. In *Proc. of EuroSys*, April 2006.

[CDH$^+$06] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of NSDI*, 2006.

[cer] CERT. `http://www.cert.org/`.

[CKK02] Yan Chen, Randy H. Katz, and John D. Kubiatowicz. SCAN: A dynamic, scalable, and efficient content distribution network. In *Proc. of Intl. Conf. on Pervasive Computing*, 2002.

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proc. of OSDI*, 1999.

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

[CML$^+$06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. of OSDI*, 2006.

[CYC$^+$01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proc. of SOSP*, 2001.

[Dan98] Peter B. Danzig. NetCache architecture and deploment. In *Computer Networks and ISDN Systems*, 1998.

[DKK$^+$01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, 2001.

[DMV03] Nikhil R. Devanur, Milena Mihail, and Vijay V. Vazirani. Strategyproof cost-sharing mechanisms for set cover and facility location games. In *Proc. of EC*, 2003.

[DW01] John R. Douceur and Roger P. Wattenhofer. Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *Proc. of MASCOTS*, 2001.

[emc] EMC Symmetrix.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. on Networking*, 8(3):281–293, 2000.

[FPT04] Alex Fabrikant, Christos H. Papadimitriou, and Kunal Talwar. The complexity of pure Nash equilibria. In *Proc. of STOC*, 2004.

[GCB$^+$02] Jim Gray, Wyman Chong, Tom Barclay, Alex Szalay, and Jan vandenBerg. TeraScale SneakerNet: Using inexpensive disks for backup, archiving, and data exchange. *Technical Report MSR-TR-2002-54*, 2002.

[GDS$^+$03] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. of SOSP*, October 2003.

[GHI$^+$01] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What can databases do for peer-to-peer? In *WebDB Workshop on Databases and the Web*, June 2001.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.

[GLMT04] Michel X. Goemans, Li Erran Li, Vahab S. Mirrokni, and Marina Thottan. Market sharing games applied to content distribution in ad-hoc networks. In *Proc. of MOBIHOC*, 2004.

[GS00] Michel X. Goemans and Martin Skutella. Cooperative facility location games. In *Proc. of SODA*, 2000.

[HAF$^+$07] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. In *Proc. of EuroSys*, 2007.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. of SOSP*, 2007.

[HL07] Galen Hunt and James Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 41(2):37–49, April 2007.

[HMD05] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, 2005.

[HS91] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology: the Journal of the International Association for Cryptologic Research*, 3(2):99–111, 1991.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[IRD02] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. of PODC*, 2002.

[jav] Java. `http://java.sun.com/`.

[JJJ$^+$00] Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. On the placement of internet instrumentation. In *Proc. of INFOCOM*, 2000.

[JJK$^+$01] Sugih Jamin, Cheng Jin, Anthony R. Kurc, Danny Raz, and Yuval Shavitt. Constrained mirror placement on the internet. In *Proc. of INFOCOM*, 2001.

[JV99] Kamal Jain and Vijay V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *Proc. of FOCS*, 1999.

[KA94] Roger M. Kieckhafer and Mohammad H. Azamanesh. Reaching approximate agreement with mixed mode faults. 3(1):53–63, January 1994.

[KAD$^+$07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. of SOSP*, 2007.

[KBC$^+$00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishnan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, November 2000.

[Kle75] Leonard Kleinrock. *Queueing Systems, Volume I: Theory*. John Wiley & Sons, January 1975.

[KP99] Elias Koutsoupias and Christos H. Papadimitriou. Worst-case equilibria. In *Proc. of STACS*, 1999.

[KR03] Bong-Jun Ko and Dan Rubenstein. A distributed, self-stabilizing protocol for placement of replicated resources in emerging networks. In *Proc. of ICNP*, 2003.

[KRS$^+$03] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. of USENIX FAST*, 2003.

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

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

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, 2001.

[LGI$^+$99] Bo Li, Mordecai J. Golin, Giuseppe F. Italiano, Xin Deng, and Kazem Sohraby. On the optimal placement of web proxies in the internet. In *Proc. of INFOCOM*, 1999.

[LKMS04] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proc. of OSDI*, 2004.

[LM07] Jinyuan Li and David Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. of NSDI*, 2007.

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

[MB02] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *Proc. of USENIX Security*, 2002.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, 1987.

[MF90] Pitu B. Mirchandani and Richard L. Francis. *Discrete Location Theory*. Wiley-Interscience Series in Discrete Mathematics and Optimization, 1990.

[MLMB01] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: Universal topology generation from a user's perspective. Technical Report 2001-003, 1 2001.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, 2002.

[mos] Mosek. `http://www.mosek.com/`.

[MP00] Ramgopal R. Mettu and C. Greg Plaxton. The online median problem. In *Proc. of FOCS*, 2000.

[MR97] Dhalia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proc. of STOC*, 1997.

[MR00] Dahlia Malkhi and Michael K. Reiter. An architecture for survivable coordination in large distributed systems. 12(2):187–202, 2000.

[MRG$^+$05] Petros Maniatis, Mema Roussopoulos, TJ Giuli, David S. H. Rosenthal, and Mary Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Trans. on Computer Systems*, 23(1):2–50, 2005.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proc. of USENIX Annual Tech. Conf.*, 1996.

[MYZ02] Mohammad Mahdian, Yinyu Ye, and Jiawei Zhang. Improved approximation algorithms for metric facility location problems. In *Proc. of Intl. Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, 2002.

[Nao91] Moni Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, 4(2):151–158, 1991.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[OR94] Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

[PST$^+$97] Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Alan Demers. Flexible update propagation for weakly consistent replication. In *Proc. of SOSP*, 1997.

[PT03] Martin Pal and Eva Tardos. Group strategyproof mechanisms via primal-dual algorithms. In *Proc. of FOCS*, 2003.

[QPV01] Lili Qiu, Venkata N. Padmanabhan, and Geoffrey M. Voelker. On the placement of web server replicas. In *Proc. of INFOCOM*, 2001.

[RCL01] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proc. of SOSP*, 2001.

[RD01] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP*, October 2001.

[RRRA99] Michael Rabinovich, Irina Rabinovich, Rajmohan Rajaraman, and Amit Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *Proc. of ICDCS*, 1999.

[s3] Amazon S3. `http://aws.amazon.com/s3/`.

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

[sfs] SFSlite. `http://www.okws.org/doku.php?id=sfslite`.

[SKKM02] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of OSDI*, 2002.

[Sri95] R. Srinivasan. RPC: Remote procedure call protocol specification version 2, 1995.

[TC02] Xueyan Tang and Samuel T. Chanson. Coordinated en-route web caching. *IEEE Trans. on Computers*, 51(6):595–607, 2002.

[tcg] TCG. `http://www.trustedcomputinggroup.org/`.

[TP88] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Proc. of SRDS*, 1988.

[Vet02] Adrian Vetta. Nash equilibria in competitive societies, with applications to facility location, traffic routing, and auctions. In *Proc. of FOCS*, 2002.

[YC07] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. In *Proc. of USENIX FAST*, 2007.

[YMV$^+$03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proc. of SOSP*, 2003.

[ZCB96] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *Proc. of INFOCOM*, 1996.

[ZSR02] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. COCA: A secure distributed online certification authority. In *ACM Trans. on Computer Systems*, 2002.