Efficient Byzantine Fault Tolerance for Scalable Storage and Services

James Hendricks

CMU-CS-09-146
July 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This material is based on research sponsored in part by the National Science Foundation, via grants CCR-0326453, DGE-0750271, DGE-0234630, CNS-0326453, CCF-0424422, and CNS-0910483, the Army Research Office, under grant number DAAD19-02-1-0389, and the Air Force Office of Scientific Research, under grant number F49620-01-1-0433 and GS00F0001P-GST004AJ0024. Support was also provided by a National Science Foundation Graduate Research Fellowship and an NDSEG Fellowship, sponsored by the Department of Defense, and the member companies of the CyLab Corporate Partners and PDL Consortium (including APC, Cisco, DataDomain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Panasas, Seagate, Sun, Symantec, and VMware).

Keywords: Distributed Systems, Distributed Storage Systems, Fault Tolerance, Reliability, Security, Byzantine Fault Tolerance, Byzantine Fault-Tolerant Storage, Homomorphic Fingerprinting, Byzantine Locking, Partial Encoding

For my beautiful wife.

Abstract

Distributed systems experience and should tolerate faults beyond simple component crashes as such systems grow in size and importance. Unfortunately, tolerating arbitrary faults, also known as Byzantine faults, poses several challenges to system designers, often limiting performance, requiring additional hardware, or both. This dissertation presents new protocols that provide substantially better performance than previously demonstrated. The Byzantine fault-tolerant erasure-coded block storage protocol proposed in this thesis provides 40% higher write throughput than the best prior approach. The Byzantine fault-tolerant replicated state machine provides a factor of 2.2–2.9 times higher throughput than the best prior approach. Furthermore, the protocols presented in this dissertation require 25–33% fewer responsive servers than the nearest competitors. To enable these results, this dissertation introduces several new techniques, including homomorphic fingerprinting, partial encoding, and Byzantine Locking, that provide unprecedented scalability, higher throughput, lower latency, and lower computational overhead. This dissertation also considers new methods for analyzing the correctness of distributed systems in the presence of faulty clients. Distributed services and storage systems built using these techniques can provide Byzantine fault tolerance in a more efficient, higher performance, and more scalable manner than previously thought possible.

Acknowledgments

I'd like to thank the members and alumni of the PDL and CyLab for their feedback, patience, and friendship, including Michael Abd-El-Malek, James Hoe, Elie Krevat, Michael Mesnier, Bryan Parno, Adrian Perrig, Brandon Salmon, Raja Sambasivan, Shafeeq Sinnamohideen, Matthew Wachs, Theodore Wong, and Jay Wylie. I'd like to acknowledge Greg for providing support throughout my tenure at Carnegie Mellon, Mike for his feedback on my proofs and arguments, and Miguel and Priya for providing feedback, insight, and support throughout my dissertation. I'd also like to thank the members and companies of the CyLab Corporate Partners and the PDL Consortium (including APC, Cisco, DataDomain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Panasas, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support.

Contents

List of Figures

Chapter 1

Introduction

As distributed systems grow in size and importance, they experience and should tolerate faults beyond simple component crashes. Distributed systems deployed in real environments experience a variety of faults, such as network misbehavior, storage failures, and software faults. Network problems include message timeouts due to temporary overload, network partitions, or packet corruption. Faulty physical storage may corrupt data or fail to make writes durable, such that a future read returns stale data. Finally, race conditions in software, drivers, or firmware may result in transient invalid results.

Though monolithic servers and simple redundancy are adequate for many applications, the largest and most critical applications must survive in ever harsher environments. Less synchronous networking delivers packets unreliably and unpredictably, and more faulty hardware and software lose data, corrupt data, and provide stale data with greater frequency. In response to this deteriorating situation, distributed protocols have experienced a natural progression over the years, from monolithic servers to replicated services, from requiring synchronous environments to allowing asynchrony, and from tolerating crashes to tolerating some corruptions through ad-hoc consistency checks. Ad-hoc consistency checks, however, may not capture important failure modes, providing the impetus for a more formalized fault model.

Protocols that can tolerate arbitrarily faulty behavior by components of the system are said to be Byzantine fault-tolerant. Ideally, systems should tolerate Byzantine faulty clients or servers. Byzantine fault tolerance ensures that all bases are covered, protecting against misdirected writes, soft errors, and other faults and corruptions found in modern hardware and software. Though Byzantine fault tolerance can protect against obscure or unlikely faults, such as a malicious insider, it is important to remember that Byzantine fault tolerance also protects against more mundane yet still perplexing faults that ad-hoc consistency checks may miss.

Unfortunately, Byzantine fault-tolerance is generally believed to be too expensive to justify in practice. This dissertation presents Byzantine fault-tolerant protocols for building storage systems and distributed services that provide much better performance and scalability than prior approaches. It develops new techniques to reduce the cost and improve the scalability and performance of Byzantine fault-tolerant distributed systems. The techniques described in this dissertation can be used to build Byzantine fault-tolerant storage systems and services that are more efficient, higher performance, and more scalable than previously thought possible.

1.1 Thesis Statement

THESIS STATEMENT: *Scalable cluster-based storage systems can tolerate Byzantine faults with substantially lower overhead than previously demonstrated. In particular, an erasure-coded Byzantine fault-tolerant block storage system can provide bandwidth on par with protocols that tolerate only crashes, and a Byzantine fault-tolerant metadata service can provide scalability and respond to most requests in a single round trip, even when only the minimal number of servers are responsive.*

To support this thesis statement, this disseration takes the following steps. First, it develops a new cryptographic primitive, which shows that erasure-coded data can be efficiently verified in a distributed system. Second, it develops a new Byzantine fault-tolerant storage protocol and proves its correctness, which showes that the performance of a Byzantine fault-tolerant block storage protocol can be competitive in theory. Third, this dissertation describes a prototype implementations of the storage protocol and competing protocols, including protocols that tolerate only crashes. It then measures and compares of the performance of the prototypes experimentally, which shows that the performance of a Byzantine fault-tolerant protocol can be competitive in practice with protocols that tolerate only crashes. Fourth, this dissertation develops a new Byzantine fault-tolerant replicated state machine protocol, proves its correctness, describes a prototype implementation, and measures the prototype against competing prototypes to demonstrate its performance and scalability properties.

Contributions: This dissertation makes four primary contributions. First, it introduces a new cryptographic primitive, homomorphic fingerprinting, which can be used to verify that distributed erasure-coded data was properly encoded. This technique allows many replication-based data storage and distribution protocols to be modified to accept erasure-coded data. Second, this dissertation proposes a correctness condition in the presence of faulty clients that allows reasoning about concurrent Byzantine-tolerant objects in terms of their sequential specification and encapsulates the semantics of the protocol in the presence of faulty clients.

Third, this dissertation introduces several techniques to improve the performance of Byzantine fault-tolerant erasure-coded storage systems, and it provides a protocol that substantially outperforms prior approaches. For example, the partial encoding optimization eliminates about half of the necessary erasure coding in a distributed storage system. A prototype implementation demonstrates experimentally that Byzantine fault-tolerant erasure-coded block storage protocols can provide similar throughput and latency as protocols that tolerate only crashes.

Fourth, this dissertation proposes a new technique for building Byzantine fault-tolerant replicated state machines, Byzantine Locking. Byzantine Locking provides unprecedented scalability and efficiency for the common case of infrequent concurrent data sharing. Byzantine Locking is used to build Zzyzx, a Byzantine fault-tolerant replicated state machine prototype that substantially outperforms prior approaches. Experiments with Zzyzx demonstrate that Byzantine fault-tolerant replicated state machines need only the minimal number of responsive servers to ensure high throughput, provide single roundtrip latency, and provide scalability through workload partitioning when the workload exhibits low object contention.

The next four sections provide an overview of these four contributions and the corresponding chapter that describes each contribution in detail. Each of the chapters also discusses the background and related work relevant to the corresponding contribution.

1.2 Verifying Distributed Erasure-Coded Data

Erasure coding can reduce the space and bandwidth overheads of redundancy in fault-tolerant data storage and delivery systems. An *m-of-n* erasure code encodes a block of data into n fragments, each $1/m^{th}$ the size of the original block, such that any m can be used to reconstruct the original block. Thus, $(n-m)$ of the fragments can be unavailable (e.g., due to corruption or server failure) without loss of access. But, it introduces the fundamental difficulty of ensuring that all erasure-coded fragments correspond to the same block of data. Without such assurance, a different block may be reconstructed from different subsets of fragments. Previous systems in which clients cannot be trusted to encode and distribute data correctly use one of two approaches. In the first approach, servers are provided the entire block of data, allowing them to agree on the contents and generate their own fragments [18, 19]. Savings are achieved for storage, but bandwidth overheads are no better than for replication. In the second approach, clients verify all n fragments when they perform a read to ensure that no other client could observe a different value [44], a significant computational overhead.

Chapter 2 proposes *homomorphic fingerprinting*, a new technique that provides this assurance without the bandwidth and computational overheads associated with current approaches. The core idea is to distribute homomorphic fingerprints with each fragment, which preserve the structure of the erasure code and allow each fragment to be independently verified as corresponding to a specific block. The key insight is that the coding scheme imposes certain algebraic constraints on the fragments, and that there exist homomorphic fingerprinting functions that preserve these constraints. Chapter 2 presents homomorphic fingerprinting functions that are secure, efficient, and compact.

1.3 Correctness in the Presence of Faulty Clients

A distributed system is correct if it faithfully implements the desired protocol specification. Unfortunately, providing correctness arguments is difficult in the presence of faulty clients that may disobey the protocol. Because faulty clients can affect the state of the system by following the protocol, their actions must be considered. But, because they may not follow the protocol, predicates needed to prove correctness may not be well defined. There are several approaches to proving correctness with faulty clients in the literature, but prior approaches prohibit some protocols that are sufficiently correct for real applications or allow protocols that may not be sufficient for some real applications.

Many Byzantine fault-tolerant protocols tolerate Byzantine faulty clients as well as servers. Most protocols are proven to guarantee some variant of linearizability [51]. In particular, in the absence of faulty clients, the execution of such protocols should result in a linearizable history of events. Unfortunately, the original definition of linearizability does not apply in the presence of faulty clients, and there is no agreed-upon definition that applies in the presence of faulty clients.

Chapter 3 proposes a minimal correctness condition, as well as two stronger conditions that allow for easier analysis and provide useful guarantees. This definition will prove useful in Chapter 4, which uses *invocation criteria*, one of the two stronger conditions, to reason about the correctness of a Byzantine fault-tolerant storage protocol.

1.4 Low-Overhead Byzantine Fault-Tolerant Storage

Unlike replicated state machine protocols, which inherently require more servers to tolerate Byzantine rather than crash faults, erasure-coded storage protocols can tolerate Byzantine faults with the same number of servers used to tolerate crashes. Given an erasure code where m fragments are required to reconstruct a block, tolerating f crash faults or Byzantine faults in an asynchronous environment requires writing fragments to $m+f$ servers (assuming $m>f$) out of $m+2f$ total servers. Real-world storage systems have recently begun to tolerate faults other than crashes, but it is unclear which faults such systems should tolerate. Thus, the Byzantine fault model would be of great interest to the distributed storage community, if shown to be sufficiently efficient.

Chapter 4 presents an erasure-coded Byzantine fault-tolerant block storage protocol that is nearly as efficient as protocols that tolerate only crashes. Previous Byzantine fault-tolerant block storage protocols have either relied upon replication, which is inefficient for large blocks of data when tolerating multiple faults, or a combination of additional servers, extra computation, and versioned storage. To avoid these expensive techniques, the protocol employs novel mechanisms to optimize for the common case when faults and concurrency are rare. In the common case, a write operation completes in two rounds of communication and a read completes in one round. The protocol requires a short checksum comprised of cryptographic hashes and homomorphic fingerprints. It achieves throughput within 10% of the crash-tolerant protocol for writes and reads in failure-free runs when configured to tolerate up to 6 faulty servers and any number of faulty clients.

1.5 Scalable Fault Tolerance through Byzantine Locking

Chapter 5 presents Zzyzx, a Byzantine fault-tolerant replicated state machine that outperforms prior approaches and provides an unprecedented feature: near-linear scaling of throughput by adding servers. Using a new technique called Byzantine Locking, Zzyzx allows a client to extract state from an underlying replicated state machine and access it via a second protocol specialized for use by a single client. This second protocol requires just one round-trip and $2f$+1 responsive servers—compared to Zyzzyva, this results in 39–43% lower response times and a factor of 2.2–2.9$\times$ higher throughput. More importantly, the extracted state can be transferred to other servers, allowing non-overlapping sets of servers to manage Zzyzx allows throughput to be scaled by adding servers when concurrent data sharing is not common. When data sharing is common, performance can match that of the underlying replicated state machine protocol (e.g., Zyzzyva).

Byzantine fault-tolerant replicated state machine protocols require $3f+1$ servers in an asynchronous network environment, of which $2f+1$ must be responsive and involved in each request. Of recent proposals, only H/Q [32] and PBFT [24] achieve this minimum, while Q/U [2] requires $4f+1$ servers to be responsive and Zyzzyva [55] requires $3f+1$ servers to be responsive. The ideal metadata protocol would requires only $2f+1$ responsive replicas, as in H/Q and PBFT, but could complete requests in as few roundtrips as possible in the common case, as in Q/U or Zyzzyva. Zzyzx achieves both goals when contention is rare, which is the case for many important workloads, such as metadata (e.g., see GPFS [92]). A detailed proof of the correctness of Byzantine Locking in general, and Zzyzx in specific, is provided in Appendix A.

Chapter 2

Homomorphic Fingerprinting: Verifying Distributed Erasure-Coded Data

Erasure coding can reduce the space and bandwidth overheads of redundancy in fault-tolerant data storage and delivery systems. An *m-of-n* erasure code encodes a block of data into n fragments, each $1/m^{th}$ the size of the original block, such that any m can be used to reconstruct the original block. Thus, $(n-m)$ of the fragments can be unavailable (e.g., due to corruption or server failure) without loss of access. Example erasure coding schemes with these properties include Reed-Solomon codes [85] and Rabin's Information Dispersal Algorithm [84].

Unfortunately, erasure coding creates a fundamental challenge: determining if a given fragment indeed corresponds to a specific original block. If this is not ensured for each fragment, then reconstructing from different subsets of fragments may result in different blocks, violating any reasonable definition of data consistency.

Systems in which clients cannot be trusted to encode and distribute data correctly use one of two approaches. In the first approach, servers are provided the entire block of data, allowing them to agree on the contents and generate their own fragments [18, 19]. Savings are achieved for storage, but bandwidth overheads are no better than for replication. In the second approach, clients verify all n fragments when they perform a read to ensure that no other client could observe a different value [44]. In this approach, each fragment is accompanied by a cross-checksum [43, 58], which consists of a hash of each of the n fragments. A reader verifies the cross-checksum by reconstructing a block from m fragments and then recomputing the other $(n-m)$ fragments and comparing their hash values to the corresponding entries in the cross-checksum, a significant computational overhead.

This chapter develops a new approach, in which each fragment is accompanied by a set of fingerprints that allows each server to independently verify that its fragment was generated from the original value. The key insight is that the coding scheme imposes certain algebraic constraints on the fragments, and that there exist homomorphic fingerprinting functions that preserve these constraints. Servers can verify the integrity of the erasure coding as evidenced by the fingerprints, agreeing upon a particular set of encoded fragments without ever needing to see them. Thus, the two common approaches described above could be used without the bandwidth or computation overheads, respectively.

The proposed fingerprinting functions belong to a family of universal hash functions [20], chosen to preserve the underlying algebraic constraints of the fragments. A particular fingerprinting function is chosen at random with respect to the fragments being fingerprinted. This "random" selection can be deterministic with the appropriate application of a cryptographic hash function [12]. If data is represented carefully, the remainder from division by a random irreducible polynomial [83] or the evaluation of a polynomial at a random point preserve the needed algebraic structure. The resulting fingerprints are secure, efficient, and compact.

The rest of this chapter is organized as follows. Section 2.1 provides a formal definition of homomorphic fingerprinting along with two such functions, the division fingerprinting and the evaluation fingerprinting functions. Section 2.2 describes a data structure called a fingerprinted cross-checksum, against which the integrity of a fragment can be verified. Section 2.3 demonstrates how homomorphic fingerprinting can improve distributed protocols by improving the bandwidth overhead of the AVID protocol [18]. Section 2.4 measures the performance of this approach, Section 2.5 considers other protocols, and Section 2.6 surveys related work.

2.1 Homomorphic Fingerprinting

This section defines homomorphic fingerprinting and its applications to erasure codes. Section 2.1.1 defines fingerprinting, providing two examples: division and evaluation fingerprinting. Section 2.1.2 defines homomorphic fingerprinting and shows that both division and evaluation fingerprinting are homomorphic fingerprinting functions. Section 2.1.3 explains the applications of homomorphic fingerprinting functions to erasure codes.

Throughout this chapter, let $\mathbb{F}$ denote a finite field with operators "$+$" and "$\cdot$", and let $\mathbb{F}_{q^k}$ denote such a field of order q^k where q is prime. Let $t \overset{R}{\leftarrow} T$ denote selection of an element from T uniformly at random and its assignment to t.

2.1.1 Fingerprinting

DEFINITION 2.1.1. An ε-*fingerprinting function* $\mathsf{fingerprint} : \mathcal{K} \times \mathbb{F}^\delta \to \mathbb{F}^\gamma$ satisfies

$$\max_{\substack{d,d' \in \mathbb{F}^\delta \\ d \neq d'}} \Pr\left[\mathsf{fingerprint}(r,d) = \mathsf{fingerprint}(r,d') : r \overset{R}{\leftarrow} \mathcal{K}\right] \leq \varepsilon$$

In words, the probability under random selection of r that $\mathsf{fingerprint}(r,d) = \mathsf{fingerprint}(r,d')$ is at most ε.

Let $\mathbb{F}_{q^k}[x]$ denote the set of polynomials with coefficients in $\mathbb{F}_{q^k}$, with "$+$" and "$\cdot$" defined as in normal polynomial arithmetic. A vector $d \in \mathbb{F}_{q^k}^\delta$ of δ elements of $\mathbb{F}_{q^k}$ has a natural representation as a polynomial $d(x) \in \mathbb{F}_{q^k}[x]$ of degree less than δ with coefficients in $\mathbb{F}_{q^k}$ where the j^{th} element of d is the coefficient in $d(x)$ of degree j, where $0 \leq j < \delta$. These notations will be used interchangeably, denoting d as $d(x)$ when it assumes this form.

EXAMPLE 2.1.2. [Rabin fingerprinting] Let $\mathbb{F}_2$ denote a field of order 2, let $\mathcal{K} = \{2, 3, 4, \ldots, 2^\gamma\}$, and let $P_2 : \mathcal{K} \to \mathbb{F}_2[x]$ be a deterministic algorithm that outputs monic irreducible polynomials of prime degree γ with coefficients in $\mathbb{F}_2$ such that

$$\Pr\left[p(x) = P_2(r) : r \overset{R}{\leftarrow} \mathcal{K}\right] = \Pr\left[p'(x) = P_2(r) : r \overset{R}{\leftarrow} \mathcal{K}\right]$$

for all $p(x), p'(x) \in \mathbb{F}_2[x]$ of degree γ. That is, P_2 selects monic degree-γ irreducible polynomials uniformly at random, where probabilities are taken with respect to the uniformly random selection of r. Rabin showed that $\mathsf{fingerprint} : \mathcal{K} \times \mathbb{F}_2^\delta \to \mathbb{F}_2^\gamma$ defined by

$$\begin{aligned}\mathsf{fingerprint}(r, d(x)) : \; & p(x) \leftarrow P_2(r); \\ & \text{return } (d(x) \bmod p(x))\end{aligned}$$

is an ε-fingerprinting function for $\varepsilon = \frac{\delta}{2^{\gamma-2}}$ [83].

THEOREM 2.1.3. [Division fingerprinting] Let $\mathbb{F}_{q^k}$ denote a field of order q^k, let the size of $\mathcal{K}$ be the number of monic irreducible polynomials of degree γ with coefficients in $\mathbb{F}_{q^k}$, and let $P_{q^k} : \mathcal{K} \to \mathbb{F}_{q^k}[x]$ be a deterministic algorithm that outputs monic irreducible polynomials of degree γ with coefficients in $\mathbb{F}_{q^k}$ chosen uniformly at random, with probabilities taken over the choice of input $r \in \mathcal{K}$ uniformly at random. Then $\mathsf{fingerprint}(r, d) : \mathcal{K} \times \mathbb{F}_{q^k}^\delta \to \mathbb{F}_{q^k}^\gamma$ defined by

$$\begin{aligned}\mathsf{fingerprint}(r, d(x)) : \; & p(x) \leftarrow P_{q^k}(r); \\ & \text{return } (d(x) \bmod p(x))\end{aligned}$$

is an ε-fingerprinting function for $\varepsilon = \frac{\delta}{q^{k\gamma} - q^{\frac{k\gamma}{2}}} \approx \frac{\delta}{q^{k\gamma}}$.

Proof. As in [83], this is because there are at least $\frac{q^{k\gamma} - q^{\frac{k\gamma}{2}}}{\gamma}$ monic degree-γ irreducible polynomials with coefficients in $\mathbb{F}_{q^k}$ [102], of which any nonzero degree-δ polynomial with coefficients in $\mathbb{F}_{q^k}$ may have at most $\lfloor \frac{\delta}{\gamma} \rfloor$ factors of degree-γ. Consider the difference of any two distinct polynomials with matching fingerprints. Let $d(x), d'(x) \in \mathbb{F}_{q^k}[x]$ and $d(x) \equiv d'(x) \bmod p(x)$ but $d(x) \neq d'(x)$. Then $(d(x) - d'(x)) \equiv 0 \bmod p(x)$, so $p(x)$ is a factor of $(d(x) - d'(x))$. Because $d(x) \neq d'(x)$, $(d(x) - d'(x)) \neq 0$. But there are at most $\frac{\delta}{\gamma}$ different monic degree-γ irreducible polynomials with coefficients in $\mathbb{F}_{q^k}$ that are factors of a nonzero degree-δ polynomial $(d(x) - d'(x))$. Hence, the probability that $p(x)$ is one of these polynomials is at most $\frac{\delta}{q^{k\gamma} - q^{\frac{k\gamma}{2}}}$. □

Division fingerprinting is a generalization of Rabin fingerprinting. Both are fast due to fast implementations of P_2 [83] and P_{q^k} [94].

Let $\mathbb{E}_{q^{k\gamma}} = \mathbb{F}_{q^k}[x]/p(x)$ denote the extension field of polynomials with coefficients in $\mathbb{F}_{q^k}$ of degree less than γ, with "$+$" defined as normal and "$\cdot$" defined modulo a constant monic degree-γ irreducible polynomial $p(x) \in \mathbb{F}_{q^k}[x]$. Let $\mathbb{E}_{q^{k\gamma}}[y]$ denote the set of polynomials with coefficients in $\mathbb{E}_{q^{k\gamma}}$, with "$+$" and "$\cdot$" defined as normal. It is convenient to consider $d \in \mathbb{E}_{q^{k\gamma}}[y]$ as a polynomial in two variables, $d(y, x)$.

A vector $d \in \mathbb{F}_{q^k}^\delta$ of δ elements of $\mathbb{F}_{q^k}$ has a natural representation as a polynomial $d(y, x) \in \mathbb{E}_{q^{k\gamma}}[y]$ of degree less than $\frac{\delta}{\gamma}$ in variable y. The j^{th} element of d is the coefficient in $d(y, x)$ of degree

$\lfloor \frac{j}{\gamma} \rfloor$ in variable y and degree j mod γ in variable x. These notations will be used interchangeably, denoting d as $d(y,x)$ when it assumes this form.

THEOREM 2.1.4. [Evaluation fingerprinting] Let $\mathbb{E}_{q^{k\gamma}} = \mathbb{F}_{q^k}[x]/p(x)$ denote a field of polynomials with coefficients in $\mathbb{F}_{q^k}$ of degree less than γ with "$\cdot$" defined modulo $p(x)$, a constant monic degree-γ irreducible polynomial. Let $\mathcal{K} = \{0,\dots,q^{k\gamma}-1\}$, and let $S : \mathcal{K} \to \mathbb{E}_{q^{k\gamma}}$ be a deterministic algorithm that outputs an element of $\mathbb{E}_{q^{k\gamma}}$ chosen uniformly at random, with probabilities taken over the choice of input $r \in \mathcal{K}$ uniformly at random. Then the function $\mathsf{fingerprint}(r,d) : \mathcal{K} \times \mathbb{F}^{\delta}_{q^k} \to \mathbb{F}^{\gamma}_{q^k}$ defined by

$$\begin{aligned} \mathsf{fingerprint}(r,d(y,x)) : \; & s(x) \leftarrow S(r); \\ & \text{return } d(s(x),x) \end{aligned}$$

is an ε-fingerprinting function for $\varepsilon = \frac{\delta/\gamma}{q^{k\gamma}}$.

Proof. As in [75], this is because any $\lceil \frac{\delta}{\gamma} \rceil$ points fully determine a polynomial of degree less than $\frac{\delta}{\gamma}$ over a field. Hence, any two distinct polynomials of degree less than $\frac{\delta}{\gamma}$ share fewer than $\frac{\delta}{\gamma}$ points. Because there are $q^{k\gamma}$ different points in $\mathbb{E}_{q^{k\gamma}}$, the probability that a randomly chosen point is shared between two distinct polynomials is at most $\frac{\delta/\gamma}{q^{k\gamma}}$. □

A trivial implementation of S is to return the polynomial representation of r divided into γ coefficients, where each coefficient is an element of $\mathbb{F}_{q^k}$.

Variants of division and evaluation fingerprinting known as the division and evaluation hashes can be used for message authentication. They are two of the fastest hashes, producing the smallest output and requiring the fewest bits of random input [79].

2.1.2 Homomorphism

Throughout this chapter, let $b \cdot d$ denote the application of "$\cdot$" by a scalar $b \in \mathbb{F}$ to each element in a vector $d \in \mathbb{F}^{\sigma}$ of σ elements of $\mathbb{F}$.

DEFINITION 2.1.5. A fingerprinting function $\mathsf{fingerprint} : \mathcal{K} \times \mathbb{F}^{\delta} \to \mathbb{F}^{\gamma}$ is *homomorphic* if $\mathsf{fingerprint}(r,d) + \mathsf{fingerprint}(r,d') = \mathsf{fingerprint}(r,d+d')$ and $b \cdot \mathsf{fingerprint}(r,d) = \mathsf{fingerprint}(r,b \cdot d)$ for any $r \in \mathcal{K}$ and any $d,d' \in \mathbb{F}^{\delta}$, $b \in \mathbb{F}$.

THEOREM 2.1.6. The fingerprinting functions given in Example 2.1.2 and Theorem 2.1.3 are homomorphic.

Proof. For any $d,d' \in \mathbb{F}^{\delta}_{q^k}$ and any $r \in \mathcal{K}$, $p(x) \leftarrow P_{q^k}(r)$,

$$\begin{aligned} \mathsf{fingerprint}(r,d(x)) + \mathsf{fingerprint}(r,d'(x)) &= d(x) \bmod p(x) + d'(x) \bmod p(x) \\ &= (d(x)+d'(x)) \bmod p(x) \\ &= \mathsf{fingerprint}(r,d(x)+d'(x)) \end{aligned}$$

Moreover, for any $b \in \mathbb{F}_{q^k}$,

$$\begin{aligned}\mathsf{fingerprint}(r, b \cdot d(x)) &= (b \cdot d(x)) \bmod p(x) \\ &= b \cdot (d(x) \bmod p(x)) \\ &= b \cdot \mathsf{fingerprint}(r, d(x))\end{aligned}$$

□

THEOREM 2.1.7. The fingerprinting function given in Theorem 2.1.4 is homomorphic.

Proof. For any $d, d' \in \mathbb{F}_{q^k}^{\delta}$ and any $r \in \mathcal{K}$, $s(x) \leftarrow S(r)$,

$$\begin{aligned}\mathsf{fingerprint}(r, d(y,x)) + \mathsf{fingerprint}(r, d'(y,x)) &= d(s(x),x) + d'(s(x),x) \\ &= (d+d')(s(x),x) \\ &= \mathsf{fingerprint}(r, d+d')\end{aligned}$$

Moreover, for any $b \in \mathbb{F}_{q^k}$,

$$\begin{aligned}\mathsf{fingerprint}(r, b \cdot d) &= \mathsf{fingerprint}(r, (b \cdot d)(y,x)) \\ &= (b \cdot d)(s(x),x) \\ &= b \cdot (d(s(x),x)) \\ &= b \cdot \mathsf{fingerprint}(r, d(y,x))\end{aligned}$$

□

The following lemma restates the properties of a homomorphic fingerprinting function.

LEMMA 2.1.8. Let $\mathsf{fingerprint} : \mathcal{K} \times \mathbb{F}^{\delta} \to \mathbb{F}^{\gamma}$ denote a homomorphic ε-fingerprinting function. For any fixed constants $b_i \in \mathbb{F}$, $1 \le i \le m$,

$$\max_{\substack{d, d_1, \ldots, d_m \in \mathbb{F}^{\delta} \\ d \neq \sum_{i=1}^{m} b_i \cdot d_i}} \Pr\left[\mathsf{fingerprint}(r, d) = \sum_{i=1}^{m} b_i \cdot \mathsf{fingerprint}(r, d_i) : r \overset{R}{\leftarrow} \mathcal{K}\right] \le \varepsilon$$

Proof. Suppose otherwise. That is, suppose that there are $d, d', d_1, \ldots, d_m \in \mathbb{F}^{\delta}$ such that $d \neq d' = \sum_{i=1}^{m} b_i \cdot d_i$ and

$$\Pr\left[\mathsf{fingerprint}(r, d) = \sum_{i=1}^{m} b_i \cdot \mathsf{fingerprint}(r, d_i) : r \overset{R}{\leftarrow} \mathcal{K}\right] > \varepsilon$$

By homomorphism, for any $r \in \mathcal{K}$,

$$\sum_{i=1}^{m} b_i \cdot \mathsf{fingerprint}(r, d_i) = \sum_{i=1}^{m} \mathsf{fingerprint}(r, b_i \cdot d_i) = \mathsf{fingerprint}(r, \sum_{i=1}^{m} b_i \cdot d_i) = \mathsf{fingerprint}(r, d')$$

Then

$$\Pr\left[\mathsf{fingerprint}(r, d) = \mathsf{fingerprint}(r, d') : r \overset{R}{\leftarrow} \mathcal{K}\right] > \varepsilon$$

in violation of Definition 2.1.1. □

Let $\mathsf{encode}(B_j)$ output

$$d_{j1} \mid \ldots \mid d_{jn}$$

Then $[\mathsf{encode}(B_1); \mathsf{encode}(B_2); \cdots ; \mathsf{encode}(B_\sigma)]$ outputs

$$\begin{array}{c|c|c} d_{11} & \ldots & d_{1n} \\ d_{21} & \ldots & d_{2n} \\ \ldots & \ldots & \ldots \\ d_{\sigma 1} & \ldots & d_{\sigma n} \end{array}$$

Let $d_i = (d_{1i}, d_{2i}, \ldots, d_{\sigma i})$. That is, each d_i is a column vector from above. Then $\mathsf{encode}^\sigma(B)$ outputs

$$d_1 \mid \ldots \mid d_n$$

Figure 2.1.1: Encoding a vector.

2.1.3 Applications to Erasure Codes

DEFINITION 2.1.9. An *m-of-n erasure coding scheme* is a pair of deterministic algorithms $(\mathsf{encode}, \mathsf{decode})$, where $\mathsf{encode} : \mathbb{F}^m \to \mathbb{F}^n$ and $\mathsf{decode} : (\mathbb{F} \times \{1, \ldots, n\})^m \to \mathbb{F}^m$. If $d_1, \ldots, d_n \leftarrow \mathsf{encode}(B)$, then $\mathsf{decode}(d_{i_1}, \ldots, d_{i_m}) = B$ for any distinct $i_1, \ldots, i_m$ ($1 \le i_j \le n$).

Each fragment provided to decode is accompanied by its index $i \in \{1, \ldots, n\}$. For notational simplicity, let each index be implicitly provided to decode.

DEFINITION 2.1.10. An m-of-n erasure coding scheme $(\mathsf{encode}, \mathsf{decode})$ is *linear* if there exist fixed constants $b_{ij} \in \mathbb{F}$ for each $1 \le i \le n$ and $1 \le j \le m$ such that for any $B \in \mathbb{F}^m$, if $d_1, \ldots, d_n \leftarrow \mathsf{encode}(B)$ then $d_i = \sum_{j=1}^m b_{ij} \cdot d_j$.

Examples of linear erasure coding schemes are Reed-Solomon codes [85] and Rabin's Information Dispersal Algorithm [84].

The following three shorthands will be useful for the next theorem. First, to consider only the i^{th} encoded fragment, define the shorthand $d_i \leftarrow \mathsf{encode}_i(B)$. Second, abbreviate $\mathsf{encode}(\mathsf{decode}(d_{i_1}, \ldots, d_{i_m}))$ as $\mathsf{encode}(d_{i_1}, \ldots, d_{i_m})$. Third, to apply encode or decode to each of the j^{th} elements of m σ-length vectors for every $j \in \{1, \ldots, \sigma\}$, define the shorthands $\mathsf{encode}^\sigma : (\mathbb{F}^\sigma)^m \to (\mathbb{F}^\sigma)^n$ and $\mathsf{decode}^\sigma : (\mathbb{F}^\sigma)^m \to (\mathbb{F}^\sigma)^m$. Then $d_1, \ldots, d_n \leftarrow \mathsf{encode}^\sigma(B)$ and $B \leftarrow \mathsf{decode}^\sigma(d_{i_1}, \ldots, d_{i_m})$, where $B \in (\mathbb{F}^\sigma)^m$ and $d_i \in \mathbb{F}^\sigma$. See Figure 2.1.1 for illustration.

THEOREM 2.1.11. Let $\mathsf{fingerprint} : \mathcal{K} \times \mathbb{F}^\delta \to \mathbb{F}^\gamma$ be a homomorphic ε-fingerprinting function, and let $(\mathsf{encode}, \mathsf{decode})$ be a linear erasure code with coefficients $b_{ij} \in \mathbb{F}$, for $1 \le i \le n$ and $1 \le j \le m$. If $(d_1, \ldots, d_n) \leftarrow \mathsf{encode}^\delta(B)$, then for any $r \in \mathcal{K}$ and any $1 \le i \le n$,

$$\mathsf{fingerprint}(r, d_i) = \mathsf{encode}_i^\gamma(\mathsf{fingerprint}(r, d_1), \ldots, \mathsf{fingerprint}(r, d_m))$$

Proof.

$$\begin{aligned}
\mathsf{fingerprint}(r, d_i) &= \mathsf{fingerprint}(r, \mathsf{encode}_i^{\delta}(B)) \\
&= \mathsf{fingerprint}(r, \sum_{j=1}^{m} b_{ij} \cdot d_j) && \text{(by Definition 2.1.10)} \\
&= \sum_{j=1}^{m} b_{ij} \cdot \mathsf{fingerprint}(r, d_j) && \text{(by Definition 2.1.5)} \\
&= \mathsf{encode}_i^{\gamma}(\mathsf{fingerprint}(r, d_1), \ldots, \mathsf{fingerprint}(r, d_m)) \\
& && \text{(by Definition 2.1.10)}
\end{aligned}$$

□

COROLLARY 2.1.12. Let $\mathsf{fingerprint} : \mathcal{K} \times \mathbb{F}^{\delta} \rightarrow \mathbb{F}^{\gamma}$ be a homomorphic ε-fingerprinting function, and let $(\mathsf{encode}, \mathsf{decode})$ be a linear erasure code. If $(d_1, \ldots, d_n) \leftarrow \mathsf{encode}^{\delta}(B)$, then for any $d \neq d_i$,

$$\Pr\left[\ \mathsf{fingerprint}(r, d) = \mathsf{encode}_i^{\gamma}(\mathsf{fingerprint}(r, d_1), \ldots, \mathsf{fingerprint}(r, d_m)) : r \xleftarrow{R} \mathcal{K}\ \right] \leq \varepsilon$$

Proof. Follows from Theorem 2.1.11 and Lemma 2.1.8. □

Theorem 2.1.11 and Corollary 2.1.12 state two useful facts about homomorphic fingerprinting functions. First, the fingerprints from an encoding of a block are equal to the encoding of the fingerprints of the block. That is, homomorphic fingerprinting functions are homomorphic. Second, if the fingerprint of a fragment is equal to the encoding of the fingerprints of other fragments, the fragment is, with high probability, the encoding of the other fragments. That is, homomorphic fingerprinting functions are fingerprinting functions.

2.2 Fingerprinted Cross-checksum

The fault-tolerant data storage example given in Section 2.3 utilizes a data structure that called a *fingerprinted cross-checksum*. Before considering the contents of a fingerprinted cross-checksum, recall the following definition of a collision-resistant hash function (e.g., see [90]).

DEFINITION 2.2.1. A family of hash functions $\{hash_K : \{0,1\}^* \rightarrow \{0,1\}^{\lambda}\}_{K \in \mathcal{K}'}$ is (τ, ε')-*collision resistant* if for every probabilistic algorithm A that runs in time τ,

$$\Pr\left[\begin{array}{r} d' \neq d \ \wedge \ hash_K(d') = hash_K(d) : \\ K \xleftarrow{R} \mathcal{K}',\ \langle d, d' \rangle \leftarrow A(K) \end{array}\right] \leq \varepsilon'$$

A fingerprinted cross-checksum then has the following form.

DEFINITION 2.2.2. An *m*-of-*n* *fingerprinted cross-checksum* fpcc consists of an array $\mathsf{fpcc.cc}[]$ of n values in $\{0,1\}^{\lambda}$ and an array $\mathsf{fpcc.fp}[]$ of m values in $\mathbb{F}^{\gamma}$.

The name "fingerprinted cross-checksum" derives from the fact that the array $\mathsf{fpcc.cc}[]$ is a cross-checksum [43, 58] and because $\mathsf{fpcc.fp}[]$ holds homomorphic fingerprints.

Let $hash : \{0,1\}^* \to \{0,1\}^\lambda$ denote a random instance of a (τ, ε')-collision resistant hash function family, and let $\mathsf{fingerprint} : \mathcal{K} \times \mathbb{F}^\delta \to \mathbb{F}^\gamma$ be a homomorphic ε-fingerprinting function. Let $\mathsf{random\_oracle} : (\{0,1\}^\lambda)^n \to \mathcal{K}$ denote a random oracle [12], which is a fixed, public function chosen uniformly at random from all functions from the same domain to the same range. The following definition specifies when a fragment is *consistent* with a fingerprinted cross-checksum.

DEFINITION 2.2.3. Let fpcc be a fingerprinted cross-checksum. A fragment $d \in \mathbb{F}^\delta$ is *consistent* with fpcc for index i, $1 \le i \le n$, if

$$\mathsf{fpcc.cc}[i] = hash(d)$$

and

$$\mathsf{fingerprint}(r, d) = \mathsf{encode}_i^\gamma(\mathsf{fpcc.fp}[1], \ldots, \mathsf{fpcc.fp}[m])$$

where $r = \mathsf{random\_oracle}(\mathsf{fpcc.cc}[1], \ldots, \mathsf{fpcc.cc}[n])$.

THEOREM 2.2.4. Let A be a probabilistic algorithm that runs in time τ, makes χ queries to $\mathsf{random\_oracle}$, and produces an m-of-n fpcc and fragments $d_{i_1}, \ldots, d_{i_m}$, and $d'_{i'_1}, \ldots, d'_{i'_m}$ such that each fragment is consistent with fpcc for its index. If

$$\begin{aligned} B &\leftarrow \mathsf{decode}^\delta(d_{i_1}, \ldots, d_{i_m}) \\ B' &\leftarrow \mathsf{decode}^\delta(d'_{i'_1}, \ldots, d'_{i'_m}) \end{aligned}$$

then the probability that $B \ne B'$ is at most $\varepsilon' + \mathcal{M} \cdot \varepsilon$ for constant $\mathcal{M} = \chi\binom{n}{m+1}$.

Proof. Suppose that A, running in time τ, produces some fpcc and fragments $d_{i_1}, \ldots, d_{i_m}$ and $d'_{i'_1}, \ldots, d'_{i'_m}$, each consistent with fpcc for its index, such that if $B \leftarrow \mathsf{decode}^\delta(d_{i_1}, \ldots, d_{i_m})$ and $B' \leftarrow \mathsf{decode}^\delta(d'_{i'_1}, \ldots, d'_{i'_m})$ then $B \ne B'$. $B \ne B'$ implies that for some j, $1 \le j \le m$, $d_{i_j} \ne \mathsf{encode}_{i_j}^\delta(B')$. Yet, because each fragment is consistent with fpcc, for each $\hat{d}_i \in \{d_{i_j}, d'_{i'_1}, \ldots, d'_{i'_m}\}$, by Definition 2.2.3

$$\mathsf{fingerprint}(r, \hat{d}_i) = \mathsf{encode}_i^\gamma(\mathsf{fpcc.fp}[1], \ldots, \mathsf{fpcc.fp}[m])$$

where $r = \mathsf{random\_oracle}(\mathsf{fpcc.cc}[1], \ldots, \mathsf{fpcc.cc}[n])$. By Definition 2.1.9, this can be rearrange to

$$\mathsf{fingerprint}(r, d_{i_j}) = \mathsf{encode}_{i_j}^\gamma(\mathsf{fingerprint}(r, d'_{i'_1}), \ldots, \mathsf{fingerprint}(r, d'_{i'_m}))$$

Bound the probability with which A succeeds in producing such values, as follows. First suppose that A fails to produce a collision in $hash$. Then, for any random oracle query $\hat{r} \leftarrow \mathsf{random\_oracle}(\mathsf{h}_1, \ldots, \mathsf{h}_n)$, A possesses at most one $\hat{d}_i$ such that $hash(\hat{d}_i) = \mathsf{h}_i$, for each $1 \le i \le n$. Of these n fragments $\hat{d}_1, \ldots, \hat{d}_n$, consider each selection of $m+1$ of them, $\hat{d}_{i_0}, \hat{d}_{i_1}, \ldots, \hat{d}_{i_m}$, such that $\hat{B} \leftarrow \mathsf{decode}^\delta(\hat{d}_{i_1}, \ldots, \hat{d}_{i_m})$ implies $\hat{d}_{i_0} \ne \mathsf{encode}_{i_0}^\delta(\hat{B})$. This selection satisfies $\mathsf{fingerprint}(\hat{r}, \hat{d}_{i_0}) = \mathsf{encode}_{i_0}^\gamma(\mathsf{fingerprint}(\hat{r}, \hat{d}_{i_1}), \ldots, \mathsf{fingerprint}(\hat{r}, \hat{d}_{i_m}))$ with probability at most ε, by Corollary 2.1.12. Since there are at most $\binom{n}{m+1}$ such selections per random oracle query, and since there are χ queries to the random oracle, the probability with which A generates any such $\hat{d}_{i_0}, \ldots, \hat{d}_{i_m}$ without finding a collision in $hash$ is at most $\mathcal{M} \cdot \varepsilon$ where $\mathcal{M} = \chi\binom{n}{m+1}$. Adding the probability ε' that A finds a collision in $hash$, the total probability of A's success is bounded by $\varepsilon' + \mathcal{M} \cdot \varepsilon$. □

2.3 Example: Improving AVID

This section illustrates how homomorphic fingerprinting can improve distributed protocols by modifying the AVID [18] protocol to make it more bandwidth efficient. Section 2.3.1 describes AVID. Section 2.3.2 highlights the proposed modifications. Section 2.3.3 provides a complete description along with pseudo-code of the modified protocol, AVID-FP. Section 2.3.4 proves that AVID-FP satisfies the functional specification of an asynchronous verifiable information dispersal protocol given in [18]. Both the AVID and AVID-FP protocols can be used to build a Byzantine fault-tolerant distributed storage system using only $3f+1$ servers [19], where f is an upper bound on the number of faulty servers.

This section assumes that there are n servers and that a data block is erasure coded into fragments such that any m fragments suffice to decode it, where $m \geq f+1$ and $n = m+2f$. The system model is similar to that in [18]; there are authenticated, reliable, asynchronous point-to-point communications channels between all servers and clients, and all servers and clients are computationally limited so as to be unable to break the utilized cryptographic primitives.

2.3.1 AVID

AVID [18] is an asynchronous verifiable information dispersal protocol. In such a protocol, a client disperses some block B, which can later be retrieved by any client. The verifiability of the protocol ensures that any two clients retrieve the same block after dispersal.

For simplicity, the description of AVID in this section is restricted to $m = f+1$ and $n = 3f+1$. To write a block, a client encodes it into fragments and computes the hash of every fragment, creating a cross-checksum. The client sends to each server its fragment and the cross-checksum. Each server then echoes the cross-checksum and its fragment to all other servers in an `echo` message. After receiving $2f+1$ fragments and matching cross-checksums in `echo` messages, a correct server decodes the block, re-encodes it, and verifies each component of the cross-checksum, aborting if inconsistencies are found. A correct server then broadcasts this consistent cross-checksum and its fragment from the re-encoding to all other servers in a `ready` message. A correct server does likewise if it receives $f+1$ `ready` messages before it receives $2f+1$ `echo` messages. After receiving $2f+1$ `ready` messages, a correct server can conclude that $f+1$ servers broadcast `ready` messages that all correct servers will eventually receive. Hence, all correct servers will broadcast `ready` messages, and so all will receive at least $2f+1$ such messages and reach this point. The server can then reconstruct its fragment if needed and store this value. The bandwidth required to store block B is then $O(n^2\frac{|\mathsf{B}|}{m}) = O(n\frac{3f+1}{f+1}|\mathsf{B}|) = O(n|B|)$, assuming the cross-checksum is of negligible size.

To read a block, a client retrieves a fragment and cross-checksum from each server until it finds a matching cross-checksum from $f+1$ servers and m fragments that are consistent with this cross-checksum. These fragments are decoded and returned.

2.3.2 AVID-FP

This section modifies AVID to utilize homomorphic fingerprinting, creating a new protocol, AVID-FP. AVID-FP differs from AVID in that servers agree upon a fingerprinted cross-checksum that is consistent with a block rather than on the block itself; servers need not echo fragments. The bandwidth required to store block B in AVID-FP is then $O(n\frac{|\mathsf{B}|}{m}) = O(\frac{m+2f}{m}|\mathsf{B}|) = O(|\mathsf{B}|)$, assuming a fingerprinted cross-checksum is of negligible size.

In AVID-FP, each cross-checksum is replaced by the fingerprinted cross-checksum from Section 2.2. Unlike a cross-checksum, a server can verify with a fingerprinted cross-checksum that its fragment corresponds to a unique block without knowing the entire block. As a consequence, there is no need to send a fragment along with each `echo` or `ready` message, which saves substantial bandwidth. Furthermore, a server has nothing to re-encode and verify upon receiving an `echo` or `ready` message, saving a substantial amount of computation.

A less welcome consequence is that a correct server cannot reconstruct its fragment if it is not provided by the client. This is not a problem, however, because a server can still verify that enough other correct servers received consistent fragments such that a consistent block will always be retrievable in the future. Hence, after a block is dispersed, at least $f+1$ correct servers will know the agreed-upon fingerprinted cross-checksum and at least m will know their fragments. To read a block, a client retrieves these $f+1$ matching fingerprinted cross-checksums and m consistent fragments.

2.3.3 AVID-FP Pseudo-code

Pseudo-code for AVID-FP can be found in Figure 2.3.2. In order to disperse a value B in AVID-FP, a client generates fragments (line 101) and the fingerprinted cross-checksum (lines 102–104). The client then sends each server its fragment and the fingerprinted cross-checksum.

Each server verifies that the fragment it receives is consistent with the fingerprinted cross-checksum (lines 600–606). If this is true, the server stores the fragment and sends an `echo` message containing the fingerprinted cross-checksum to all other servers (lines 607–608).

Upon receiving $m+f$ `echo` messages with matching fingerprinted cross-checksums from unique servers, a server can determine that at least m correct servers sent such messages and hence stored fragments consistent with the fingerprinted cross-checksum (line 701). The server then sends a `ready` message containing the fingerprinted cross-checksum to all other servers (line 702).

If a server receives $f+1$ `ready` messages with matching fingerprinted cross-checksums from unique servers (line 801), at least one must be from a correct server that determined that at least m correct servers stored consistent fragments. Hence, such a server can determine likewise and send a `ready` message to all other servers, if it has yet to do so (line 802).

If a server receives $2f+1$ `ready` messages with matching fingerprinted cross-checksums from unique servers, at least $f+1$ must be from correct servers (line 804). Hence, each correct server will receive at least these $f+1$ matching `ready` messages. Then each correct server will send a `ready` message (lines 801–802), so each correct server will actually receive at least $2f+1$ matching `ready` messages. Thus, a correct server can conclude upon receiving $2f+1$ `ready` messages that all correct servers will eventually receive $2f+1$ `ready` messages, as well. The server can then save the agreed upon fingerprinted cross-checksum and respond to the client (line 806).

Upon receiving $2f+1$ responses, the client is assured that $f+1$ correct servers have saved the same fingerprinted cross-checksums and that m correct servers have stored fragments consistent with this fingerprinted cross-checksum. To retrieve a block, then, a client retrieves a fragment and fingerprinted cross-checksum from each server, waiting for matching fingerprinted cross-checksums

```
c_disperse(B):                                  /* Client disperse protocol */
100: $\mathsf{store\_count} \leftarrow 0$
101: $\mathsf{d}_1,\ldots,\mathsf{d}_n \leftarrow \mathsf{encode}^{\delta}(\mathsf{B})$
102: for $(i \in \{1,\ldots,n\})$ do $\mathsf{fpcc.cc}[i] \leftarrow \mathsf{hash}(\mathsf{d}_i)$
103: $r \leftarrow \mathsf{random\_oracle}(\mathsf{fpcc.cc}[1],\ldots,\mathsf{fpcc.cc}[n])$
104: for $(i \in \{1,\ldots,m\})$ do $\mathsf{fpcc.fp}[i] \leftarrow \mathsf{fingerprint}(r,\mathsf{d}_i)$
105: for $(i \in \{1,\ldots,n\})$ do send(disperse, fpcc, $\mathsf{d}_i$) to $\mathsf{S}_i$

Upon receiving (stored) from $\mathsf{S}_i$ for the first time
200: $\mathsf{store\_count} \leftarrow \mathsf{store\_count}+1$
201: if $(\mathsf{store\_count} = 2f+1)$ then return SUCCESS

c_retrieve():                                   /* Client retrieve protocol */
300: $\mathsf{fpcc} \leftarrow$ NULL;   $State[*] \leftarrow \langle$NULL, NULL, NULL$\rangle$
301: for $(i \in \{1,\ldots,n\})$ do send(retrieve) to $\mathsf{S}_i$

Upon receiving (retrieved, $\widehat{\mathsf{fpcc}}$, $\langle \mathsf{fpcc}', \mathsf{d}\rangle$) from $\mathsf{S}_i$
400: if $(\widehat{\mathsf{fpcc}} \neq$ NULL$)$ then
401:     $State[i] \leftarrow \langle \widehat{\mathsf{fpcc}}$, NULL, NULL$\rangle$          /* Potential fpcc */
402:     if $(|\{j : State[j] = \langle \widehat{\mathsf{fpcc}},*,*\rangle\}| = f+1)$ then
403:         $\mathsf{fpcc} \leftarrow \widehat{\mathsf{fpcc}}$                           /* Found fpcc */
404:
405: if (NULL $\neq \mathsf{fpcc}'$) then
406:     $\mathsf{h} \leftarrow \mathsf{hash}(\mathsf{d})$
407:     $r \leftarrow \mathsf{random\_oracle}(\mathsf{fpcc}'.\mathsf{cc}[1],\ldots,\mathsf{fpcc}'.\mathsf{cc}[n])$
408:     $\mathsf{fp} \leftarrow \mathsf{fingerprint}(r,\mathsf{d})$
409:     $\mathsf{fp}' \leftarrow \mathsf{encode}_i^{\gamma}(\mathsf{fpcc}'.\mathsf{fp}[1],\ldots,\mathsf{fpcc}'.\mathsf{fp}[m])$
410:     if $(\mathsf{fp} = \mathsf{fp}' \wedge \mathsf{h} = \mathsf{fpcc}'.\mathsf{cc}[i])$ then
411:         $State[i] \leftarrow \langle \widehat{\mathsf{fpcc}}, \mathsf{fpcc}', \mathsf{d}\rangle$   /* Consistent fragment */
412:
413: if $(\mathsf{fpcc} \neq$ NULL$)$ then
414:     $Frags \leftarrow \{\mathsf{d}_j : State[j] = \langle *, \mathsf{fpcc}, \mathsf{d}_j\rangle\}$
415:     if $(|Frags| = m)$ then return $\mathsf{decode}^{\delta}(Frags)$
```

```
s_init():                                       /* Initialize server state */
500: $\mathsf{echoed} \leftarrow \langle$NULL, NULL$\rangle$;   $\mathsf{verified} \leftarrow$ NULL
501: $EchoSet_* \leftarrow \emptyset$;   $ReadySet_* \leftarrow \emptyset$

/* Server $i$ code to disperse data */
Upon receiving (disperse, fpcc, $\mathsf{d}_i$) from client
600: $\mathsf{h} \leftarrow \mathsf{hash}(\mathsf{d}_i)$
601: $\mathsf{h}' \leftarrow \mathsf{fpcc.cc}[i]$
602: $r \leftarrow \mathsf{random\_oracle}(\mathsf{fpcc.cc}[1],\ldots,\mathsf{fpcc.cc}[n])$
603: $\mathsf{fp} \leftarrow \mathsf{fingerprint}(r,\mathsf{d}_i)$
604: $\mathsf{fp}' \leftarrow \mathsf{encode}_i^{\gamma}(\mathsf{fpcc.fp}[1],\ldots,\mathsf{fpcc.fp}[m])$
605:
606: if $(\mathsf{echoed} = \langle$NULL, NULL$\rangle \wedge \mathsf{fp} = \mathsf{fp}' \wedge \mathsf{h} = \mathsf{h}')$ then
607:     $\mathsf{echoed} \leftarrow \langle \mathsf{fpcc}, \mathsf{d}_i\rangle$
608:     for $(j \in \{1,\ldots,n\})$ do send(echo, fpcc) to $\mathsf{S}_j$

Upon receiving (echo, fpcc) from $\mathsf{S}_j$
700: $EchoSet_{\mathsf{fpcc}} \leftarrow EchoSet_{\mathsf{fpcc}} \cup \{j\}$
701: if $(|EchoSet_{\mathsf{fpcc}}| = m+f \wedge |ReadySet_{\mathsf{fpcc}}| < f+1)$ then
702:     for $(j \in \{1,\ldots,n\})$ do send(ready, fpcc) to $\mathsf{S}_j$

Upon receiving (ready, fpcc) from $\mathsf{S}_j$
800: $ReadySet_{\mathsf{fpcc}} \leftarrow ReadySet_{\mathsf{fpcc}} \cup \{j\}$
801: if $(|ReadySet_{\mathsf{fpcc}}| = f+1 \wedge |EchoSet_{\mathsf{fpcc}}| < m+f)$ then
802:     for $(j \in \{1,\ldots,n\})$ do send(ready, fpcc) to $\mathsf{S}_j$
803:
804: if $(|ReadySet_{\mathsf{fpcc}}| = 2f+1)$ then
805:     $\mathsf{verified} \leftarrow \mathsf{fpcc}$
806:     send(stored) to client

/* Server $i$ code to retrieve data */
Upon receiving (retrieve) from client
900: send(retrieved, verified, echoed) to client
```

Figure 2.3.2: AVID-FP pseudo-code

from $f+1$ servers (lines 402–403) and consistent fragments from m servers (line 415). These fragments are then decoded and the resulting block is returned.

2.3.4 AVID-FP Correctness

To see why this is correct, recall the definition of an asynchronous verifiable information dispersal scheme given in [18]:

DEFINITION 2.3.1. An (m,n)-*asynchronous verifiable information dispersal scheme* is a pair of protocols (disperse, retrieve) that satisfy the following with high probability:

Termination: If disperse(B) is initiated by a correct client, then disperse(B) is eventually completed by all correct servers.

Agreement: If some correct server completes disperse(B), all correct servers eventually complete disperse(B).

Availability: If $f+1$ correct servers complete disperse(B), a correct client that initiates retrieve() eventually reconstructs some block B'.

Correctness: After $f+1$ correct servers complete disperse(B), all correct clients that initiate retrieve() eventually retrieve the same block B'. If the client that initiated disperse(B) was correct, then $\mathsf{B}' = \mathsf{B}$.

Termination is simple, as in the original AVID protocol. If a correct client initiates disperse, it erasure codes the block and computes a valid fingerprinted cross-checksum before dispersing fragments to each server (lines 101–105). Eventually, at least $m+f$ correct servers receive `disperse` messages, verify their fragments against the fingerprinted cross-checksum, and send `echo` messages to all other servers (line 608). Each correct server eventually receives at least $m+f$ `echo` messages; it will then send a `ready` message (line 702) unless it has already done so (line 802). Thus each correct server will eventually receive at least $2f+1$ `ready` messages, at which point it will send a `stored` message to the client and complete. Hence, all correct servers eventually complete.

Agreement is simpler than in the original AVID protocol because a server in AVID-FP need not reconstruct the block before returning a `ready` message. If some correct server completes disperse(B), then it received $2f+1$ `ready` messages (line 804). At least $f+1$ must have come from correct servers, so all correct servers will eventually receive `ready` messages from these servers. Then the condition satisfied on either line 801 or line 701 will be met for all correct servers, so all correct servers will send `ready` messages and receive at least $2f+1$ such messages, thus completing.

Availability is different than in the original AVID protocol. In AVID, fragments must be echoed such that a correct server can reconstruct its fragment if needed; in AVID-FP, fragments are not echoed. If any correct server completes disperse, it received $2f+1$ `ready` messages. Then at least one correct server received $m+f$ `echo` messages. If not, at most f `ready` messages would be received by any correct server, because no correct server would meet the condition on line 701. Hence, at least m correct servers stored consistent fragments (line 607). Then after $f+1$ correct servers complete disperse, a client that initiates retrieve will eventually receive $f+1$ matching fingerprinted cross-checksums (saved on line 805) along with m consistent fragments, which it will decode and return as some block B'.

Correctness is similar to the original AVID protocol except that the properties of the homomorphic fingerprint are required. Suppose some correct server saves fpcc_1 on line 805 and some other correct server saves $\mathsf{fpcc}_2 \neq \mathsf{fpcc}_1$. Then $m+f$ servers echoed fpcc_1, of which at least m were correct, and $m+f$ servers echoed fpcc_2, of which at least m were correct. Because a correct server will only echo once (line 606 will never be satisfied after line 607 is reached), there are at least $m+m+f$ servers involved, which is a contradiction (there are only $n < m+m+f$ servers in the system). Hence, any block decoded during retrieve is consistent with the same fpcc. Furthermore, if a correct client initiated disperse(B), this fpcc will be consistent with B. Then, by Theorem 2.2.4, the probability that $\mathsf{B} \neq \mathsf{B}'$ is negligible, for appropriately chosen parameters.

2.4 Performance

Homomorphic fingerprinting is efficient, contributing little overhead to distributed protocols. To demonstrate that homomorphic fingerprinting is not a substantial computational burden in protocols such as the AVID-FP protocol given above, this section compares an implementation of the evaluation fingerprinting function against cryptographic hashing. The evaluation fingerprinting function implementation in this section is similar to the evaluation hash considered in [79] and [94].

A polynomial
$$d(y,x) = a_\sigma(x) \cdot y^\sigma + \ldots + a_0(x) \cdot y^0 \in \mathbb{E}_{q^{k\gamma}}[y]$$

can be evaluated using Horner's rule. To do so, let $\mathsf{fp} \leftarrow 0$, and for $j = \sigma, \ldots, 0$, iteratively compute $\mathsf{fp} \leftarrow \mathsf{fp} \cdot y + a_j(x)$. The efficiency of this implementation then depends on an efficient implementation of "$+$" and "$\cdot$" for $a_j(x), y \in \mathbb{E}_{q^{k\gamma}} = \mathbb{F}_{q^k}[x]/p(x)$, where y, the point at which to evaluate, is the fixed random value $s(x) \leftarrow S(r)$.

Given an implementation of "$+$" and "$\cdot$" for $\mathbb{F}_{q^k}$, construct "$+$" and "$\cdot$" for $\mathbb{F}_{q^k}[x]/p(x)$ as follows. Consider the representation of $a(x) \in \mathbb{F}_{q^k}[x]/p(x)$ as a polynomial

$$a(x) = b_{\gamma-1} \cdot x^{\gamma-1} + \ldots + b_0$$

where $b_i \in \mathbb{F}_{q^k}$. The "$+$" operator is defined as the addition of same-degree terms. The "$\cdot$" operator is defined as multiplication of two polynomials of degree less than γ modulo a constant monic degree-γ irreducible polynomial $p(x) \in \mathbb{F}_{q^k}[x]$.

For fixed $s(x)$, compute $a(x) \cdot s(x)$ as follows. For $0 \le i < \gamma$, build γ lookup tables mapping each $b_i \in \mathbb{F}_{q^k}$ to $b_i \cdot x^i \cdot s(x) \bmod p(x)$; that is, compute the map $b_i \mapsto b_i \cdot x^i \cdot s(x) \bmod p(x)$. Each of these γ tables will contain q^k entries that are each $\lceil \log_2 q^{k\gamma} \rceil$ bits wide. A 128-bit fingerprint over $\mathbb{F}_{2^8}$ must compute 16 such tables after the random value r is selected; each table is 4 kB, for a total of 64 kB. Given these tables, one can compute $a(x) \cdot s(x)$ as the sum of γ lookups, $\Sigma_{i=0}^{\gamma-1}(b_i \cdot x^i \cdot s(x) \bmod p(x))$. For $\mathbb{F}_{2^8}$, this requires a table lookup plus an exclusive-or per byte of input.

For $\mathbb{F}_{2^8}$, building these tables is efficient: "$+$" is simply exclusive-or, and "$\cdot$" can be implemented using a 64 kB lookup table. The "mod" operator can be defined using "$+$" and "$\cdot$". Because $p(x)$ is constant, "mod" can be implemented with a lookup table for $b_i \mapsto b_i \cdot x^\gamma \bmod p(x)$. This table will contain 2^8 entries of γ bytes each, for a total of 4 kB for a 128-bit fingerprint, and it can be computed before the random value r is selected.

Gladman's implementation of SHA-1 [42] achieves a throughput of 110 megabytes per second on a 3 GHz Intel Pentium D. On this machine, the time to compute lookup tables for the evaluation fingerprint implementation presented here is 20 microseconds. After this computation, this implementation achieves a throughput of 410 megabytes per second.

2.5 Other Protocols

m-of-n erasure coding is used in many distributed systems (e.g., [5, 19, 22, 44, 62, 91]), because it reduces storage, network bandwidth, and I/O bandwidth. The savings approaches a factor of m when compared to replication. The division and evaluation fingerprinting functions are homomorphic over several popular erasure codes. Reed-Solomon codes [85] interpolate a polynomial over a field $\mathbb{F}_{q^k}$, and Rabin's Information Dispersal Algorithm [84] encodes using an $n \times m$ matrix over a field $\mathbb{F}_{q^k}$ where every $m \times m$ submatrix is invertible. Both are linear erasure codes over $\mathbb{F}_{q^k}$. A common field is $\mathbb{F}_{2^8}$ such that field elements are bytes. Rabin fingerprinting is homomorphic over many erasure codes based solely on exclusive-or, such as Online Codes [74] and parity.

Homomorphic fingerprinting provides benefits to erasure-coded Byzantine fault-tolerant storage systems [19, 44]. Section 2.3 demonstrated how the AVID protocol [18], used in [19], can exploit homomorphic fingerprinting to be more bandwidth efficient. Variants of the PASIS protocol [44, 45] can also exploit homomorphic fingerprinting. In the "non-repairable" protocol a writer sends fragments along with a cross-checksum to each server; a reader returns a block after finding sufficient servers with fragments and matching cross-checksums. Before accepting a value, a reader must reconstruct all fragments and recompute the cross-checksum, a significant computational overhead. This protocol can benefit directly by replacing the cross-checksum with a fingerprinted cross-checksum, obviating the need for fragment reconstruction and cross-checksum recomputation. The "repairable" protocol can also benefit, but requires further modifications.

Homomorphic fingerprinting may also provide benefits to erasure-coded broadcast [22], content distribution [61], and similar applications, if the encoding is not trusted to be consistent without verification.

2.6 Related Work

A common cryptographic application of universal hashing is for message authentication codes (MACs) [79]. An early proposal by Krawczyk [59] included a MAC similar to Rabin's fingerprints. Shoup presented faster variants [94] along with implementation suggestions to optimize performance. Nevelsteen compares several other variants [79].

Homomorphic fingerprinting functions share homomorphic properties with incremental hashing functions [11]. Incremental hashing, however, is substantially slower because it is based on number-theoretic primitives. The homomorphic properties of incremental hashing are exploited in [61], which applies these homomorphic properties to Online Codes [74] in a peer-to-peer content distribution network.

The algebraic properties of certain universal hashes has been examined before. Rabin used these properties to update the fingerprint of a file [83]. In [93], a similar technique is used by a disk scrubber to check the consistency of erasure-coded data in a benign environment. In [15], algebraic properties are leveraged to permit fast updates of Rabin fingerprints of data structures such as trees.

More distantly related to this technique is verifiable secret sharing (e.g., [29, 39, 80, 98]), which allows correct participants to verify that a secret was shared among them consistently. The secrecy of the shared value, however, which must be preserved throughout the share distribution and verification process, drives these protocols to employ number-theoretic techniques that are significantly heavier-weight than considered here.

It is worth mentioning that a random oracle, as in Section 2.2, can be replaced with an evaluation of a distributed pseudo-random function [78] in a protocol such as AVID-FP. This construction has the benefit of requiring only standard cryptographic assumptions.

2.7 Conclusion

Homomorphic fingerprinting enables efficient verification that fragments have been correctly generated by an erasure-coding of a particular data block. A high level of security can be achieved with small fingerprints, and fingerprint generation has lower computational overhead than cryptographic hashing. This technique provides benefits to several distributed protocols. In particular, distributed storage systems capable of tolerating Byzantine clients, which may attempt to write sets of fragments that reconstruct different values depending upon which subset is used, can benefit significantly from this mechanism.

Chapter 3

The Correctness of Distributed Systems in the Presence of Faulty Clients

As the complexity and significance of distributed systems increases, ensuring that such systems operate as expected becomes both more difficult and more important. Thus, proposals for systems that depend on complex distributed protocols are often accompanied by arguments that such protocols remain live under appropriate conditions and implement the expected operational semantics. Such arguments are called *liveness* and *safety* arguments. There are several properties that a client-server protocol may ensure to demonstrate liveness (e.g., *wait-freedom* [49] or *obstruction-freedom* [50]). When the protocol model is a set of servers providing clients access to a concurrent object, safety is often described in terms of *linearizability* [51], which allows a concurrent object to be reasoned about in terms of its sequential specification.

As distributed systems grow in size and importance, they must also tolerate faults other than crashes. Unfortunately, safety conditions such as linearizability do not apply when clients may disobey the protocol specification because predicates needed to demonstrate such conditions may not be well defined. Linearizability relates potential real-time precedence to the invocations and responses of operations as issued and seen by clients, but a faulty client may not properly invoke an operation. Though a protocol need not provide guarantees about responses provided to faulty clients, it must account for the effect of invocations by faulty clients. After all, a faulty client can always invoke an operation, if only by following the protocol correctly.

This chapter considers the correctness of distributed systems that tolerate *Byzantine-faulty* clients, which may exhibit arbitrary or even malicious behavior, and require a safety condition similar to linearizability. Byzantine fault tolerance [65] has become increasingly important in the distributed systems community. Safety conditions similar to linearizability define the correct behavior of Byzantine fault-tolerant systems such as replicated state machines [2, 24, 32, 55] and storage systems [19, 44, 48, 66, 68].

Of course, definitions of safety in the presence of faulty clients have been previously considered; after all, one must define correctness before arguing that a protocol is correct. Two common approaches are to extend linearizability to apply in the presence of faulty clients and to define a new safety condition, possibly arguing its similarity to linearizability. Section 3.3.2 considers prior extensions to linearizability, but prior extensions may not ensure the expected operational semantics for some applications, and prior extensions may preclude some protocols that provide operational semantics that suffice for many applications. Defining a new safety condition is even worse, as it obfuscates the operational semantics of a protocol. A non-standard safety condition complicates protocol comparison, obscures which faults are tolerated, and may fail to ensure the expected operational semantics.

This chapter proposes a minimal correctness condition that is restrictive enough to prevent anomalies but permissive enough to allow all prior protocols. For applications that require stricter semantics, this chapter provides a mechanism to strengthen this condition. By separating basic correctness requirements from additional protocol features, this chapter strives to facilitate comparison of protocol guarantees. Furthermore, explicit delineation of protocol features ensures that a protocol provides adequate operational semantics when deployed in a real system without unduly restricting the design of the protocol.

This rest of this chapter is organized as follows. Section 3.2 presents a formal definition of linearizability in the presence of faulty clients that does not preclude previous protocols that appear correct to correct clients. Section 3.3 provides techniques to ensure stricter operational semantics as needed, but notes that such semantics are not always possible. In particular, Section 3.3.2 defines *immediate recovery*, which ensures a faulty client cannot affect a block storage or atomic register protocol after the client has been revoked. Section 3.3.2 then proves that entirely wait-free protocols cannot provide immediate recovery. In contrast to Section 3.3.2, Sections 3.4 presents the first block storage protocol that provides wait-free reads and writes and ensures *immediate recovery*. By necessity, revocation is not wait-free.

3.1 Background

This chapter is concerned with distributed systems in which a set of clients perform concurrent operations on objects controlled by a set of servers. The standard definition of safety for such distributed systems is linearizability as defined by Herlihy and Wing [51], which defines safety from the perspective of correct clients. For reference, this chapter restates the definition of linearizability in Section 3.2 as Definition 3.2.2. Proofs of safety ensure that a protocol adheres to a particular specification. This chapter argues that previous attempts to define safety conflate safety with additional protocol features. Additional features may be necessary for a particular application, but are not required to ensure safety in the traditional sense. Section 3.3 describes features that may be useful but go beyond safety.

When might traditional notions of safety be insufficient? Consider an accounting firm with a shared distributed storage system in an asynchronous environment. As is necessary in many practical storage protocols, the read subprotocol includes a *write back* step, in which the client writes the block read back into the system to ensure subsequent reads are up-to-date [28]. A faulty client—for example, a computer operated by a malicious employee that is subsequently fired—may partially complete a write, such that a subsequent read operation may or may not return the value written. (For example, PASIS would classify the read as either *repairable* or *incomplete* [44].) For many days, weeks, or years, reading the block may not return the partially-written value. But, at some distant point in the future, a different subset of servers may cause a client to read and hence write back the partial write, wreaking havoc. For example, the faulty client could overwrite all blocks with zeroes, with the consequence only discovered long after the client has left the system.

The Byzantine fault-tolerant storage community has struggled with this problem for several years. Malkhi, Reiter, and Lynch [70] extended the concept of linearizability to distributed storage systems with Byzantine-faulty clients. Their proposal requires that the number of operations issued by faulty clients is finite if all faulty clients eventually leave the system. A notable drawback of this approach is that it only applies to infinite executions of a system, because all clients issue finite operations in finite executions, system executions are finite in practice. Liskov and Rodrigues [66] defined two safety conditions for Byzantine fault-tolerant storage, BFT-linearizable and BFT-linearizable+. Unfortunately, several reasonable storage protocols meet neither safety condition, and neither condition precludes certain anomalies, such as a write from a faulty client taking effect years after the client was removed from the system. The benefits and drawbacks of these extensions are considered in Section 3.3.2. Aguilera and Swaminathan described a property called *limited effect*, which is similar in intention to immediate recovery, but they achieve their property by implementing a weaker primitive (an abortable register) to allow wait-freedom [6].

There have been several recent protocols that tolerate faulty clients in some form. Castro and Liskov [24], Abd-el-Malek et al. [2], Cowling et al. [32], and Kotla et al. [55] all consider Byzantine-faulty clients in replicated state machines. Malkhi and Reiter [68], Goodson et al. [44], Liskov and Rodrigues [66], Cachin and Tessaro [19], and Hendricks et al. [48] all consider Byzantine-faulty clients in storage systems. Because there is no agreed upon definition of correctness in the presence of faulty clients, protocol designers often create their own definition. Choosing the right definition of correctness in such an ad-hoc manner is difficult if not perilous, and one could easily imagine the process going awry. To state the problem another way, using formal methods to prove ad-hoc criteria burdens the protocol designer without ensuring traditional notions of correctness.

3.2 Safety in the Presence of Faulty Clients

This section proposes the minimal requirements for safety. Informally, for a protocol to be correct in the presence of faulty clients, from the perspective of the correct clients, any execution of the protocol could have happened if all clients were correct. Before restating this more formally, define a few terms following the terminology of Herlihy and Wing [51, Section 2.1]. A *client* (or process) issues operations in sequence. An operation consists of two *events*: an *invocation* and a matching *response*. The execution of a distributed system is modeled by a *history*, which is an ordered sequence of invocation and response events by clients. Each event is associated with the client that invoked the operation. A protocol is said to be *correct* if any execution of the protocol will result in a history that satisfies the *correctness condition*. Consider the following definition of an *extension* that accounts for faulty clients to a correctness condition that does not:

DEFINITION 3.2.1. A history H comprised of events from all correct clients satisfies a faulty-client *extension* to a correctness condition if and only if there exists history $\hat{H}$ such that

1. $\hat{H}$ satisfies the correctness condition and
2. The subsequence of $\hat{H}$ consisting of each event from every correct client is equal to H.

This definition satisfies two important properties. First, in the absence of faulty clients, a protocol will satisfy the traditional notion of correctness. This is because if all clients are correct, H is identical to $\hat{H}$ and thus H satisfies the correctness condition. Thus, an extension to linearizability will imply linearizability in the absence of faulty clients. Second, this definition prevents anomalies. For example, a state machine protocol cannot transition to an unreachable state from the perspective of correct clients. An execution of a protocol is only correct if, from the perspective of each correct client, the execution could have resulted from a set of clients correctly executing the protocol. In other words, any effects of faulty clients can be explained away as though such faulty clients were correct.

For a particular correctness condition such as linearizability in the presence of Byzantine-faulty clients, Definition 3.2.1 can be specialized as a *Byzantine-faulty-client extension of linearizability*.[1] Linearizability relates the *sequential specification* of a protocol to the correctness of its distributed variant. The benefit of linearizability is that the sequential specification of a protocol is often simpler than its distributed equivalent because operations may not be totally ordered in a distributed system; for example, the sequential specification of a storage register is that a read returns the current value and a write sets the value to the argument of the write.

Before continuing, define a few more terms following Herlihy and Wing [51]. Two histories are *equivalent* if, for each client, the ordered subsequences of events associated with that client from either history are the same. A history is *sequential* if the first event is an invocation; every invocation, except possibly the last, is followed by a matching response; and every response is preceded by a matching invocation. A legal sequential history is a sequential history that conforms to the sequential specification. A history H can be *extended* by including a matching response for some of its unmatched invocations, and it can be *completed* by omitting unmatched invocations. Recall the definition of a linearizable history [51, Section 2.2]:

DEFINITION 3.2.2. A history H is *linearizable* if

1. H can be extended to some history H' such that complete(H') is equivalent to some legal sequential history S and
2. for each response that precedes some invocation in H, the response also precedes that invocation in S.

A Byzantine extension to linearizability is then defined as follows:

DEFINITION 3.2.3. A history H comprised of events from all correct clients satisfies a *Byzantine-faulty-client extension to linearizability* if there exists some history $\hat{H}$ such that

1. $\hat{H}$ can be extended to some history H' such that complete(H') is equivalent to some legal sequential history S,
2. For each response that precedes some invocation in H, the response also precedes that invocation in S, and
3. The subsequence of $\hat{H}$ consisting of each event from every correct client is equal to H.

Definition 3.2.3 is a direct application of Definition 3.2.1 to Definition 3.2.2, making Definition 3.2.3 the natural extension of linearizability in the presence of faulty clients. The first and second terms

[1]Or a Byzantine extension of linearizability, or just Byzantine linearizability, for short.

restate Definition 3.2.2. The first term corresponds to Definition 3.2.1. The second term is simplified by using H rather than $\hat{H}$. The second term can use H rather than $\hat{H}$ because any $\hat{H}$ that satisfies the first and third term would still satisfy these terms if reordered such that responses to faulty clients do not precede any invocations and invocations by faulty clients precede all responses. Thus, invocations and responses from faulty clients need not apply to the second term, so H is equivalent to $\hat{H}$ for this purpose.

3.3 Stricter Extensions of Linearizability

A protocol that tolerates faulty clients must ensure that all protocol executions satisfy Definition 3.2.3, but Definition 3.2.3 may not ensure sufficient operational semantics for some applications. This section considers additional protocol features that restrict the ability of faulty clients to invoke operations. Protocol designers should describe such features when specifying what semantics a protocol implementation provides. This section considers invocation criteria and recovery. Most protocols should at least consider invocation criteria, which describes when an invocation from a faulty (or correct) client is allowed to appear in a history.

3.3.1 Invocation Criteria

Recall that history $\hat{H}$ in Definition 3.2.3 includes events associated with both correct and faulty clients. A faulty-client invocation criteria defines the conditions under which invocations from faulty clients may be present into $\hat{H}$. By including invocation criteria when describing protocol semantics, the effects of faulty clients can be made obvious and application designers can determine if a protocol provides acceptable semantics.

DEFINITION 3.3.1. The *faulty-client invocation criteria* is a set of requirements that must be satisfied for each invocation associated with a faulty client.

A useful invocation criteria is that a faulty client must successfully complete a remote procedure call at a correct server for each invocation in $\hat{H}$ associated with that client. This ensures that a faulty client must perform a particular action in order to affect the state of the system. Specifying invocation criteria is a useful sanity check in the design of storage protocols, and allows application designers to choose an appropriate protocol.

3.3.2 Recovery

The notion of *recovery* from faulty clients has been proposed as a desirable or necessary property of storage protocols that tolerate Byzantine-faulty clients. Recovery can also apply to non-storage protocols. Recovery is usually described in relation to a special STOP event in an execution history. After a STOP event is issued for a client, the client can issue only a limited number of operations. In practice, a STOP event is issued when a faulty client is repaired or removed from the system, perhaps after being detected as faulty. The least restrictive variant of recovery is eventual recovery.

DEFINITION 3.3.2. A distributed protocol *eventually recovers* from faulty clients if each faulty client issues only a finite number of operations after a STOP event is issued for all faulty clients.

Eventual recovery was proposed by Malkhi et al. [70] as “Byznearizability.” It ensures that faulty clients cannot affect the state of the system after all faulty clients are removed and enough time has passed. Unfortunately, eventual recovery provides few guarantees in practice—any finite history trivially satisfies eventual recovery, and all histories encountered in practice are, of course, finite.

Despite this criticism, demonstrating that a protocol eventually recovers is useful because protocols that do not eventually recover exhibit strange semantics: even after removal from the system, faulty clients continuously change the state of the system. Eventual recovery is often implied by a reasonable faulty-client invocation criteria. For example, if a client must communicate with a correct server to invoke an operation, the protocol will eventually recover from faulty clients. A slightly more restrictive property is bounded recovery:

DEFINITION 3.3.3. A distributed protocol *recovers after bounded operations* if for some finite constant b, after a STOP event is issued for a faulty client the client issues at most b operations.

Bounded recovery was proposed by Liskov and Rodrigues [66] as “BFT-linearizability.” Liskov and Rodrigues further refined the definition to ensure that operations cannot be issued by a faulty client after a STOP event is issued for the client and the state of the system is overwritten k times. They call this stricter definition “BFT-linearizability+.” Storage protocols that satisfy BFT-linearizability or BFT-linearizability+ bound the number of “lurking writes” [66], which are writes by a client that occur after the client has left the system.

Unfortunately, bounded recovery may be too restrictive—most recent storage protocols do not ensure bounded recovery with no apparent ill consequences [19, 44, 48, 68]. Augmenting a storage protocol to ensure bounded recovery may be a sizable undertaking. Furthermore, for large values of b, faulty clients can issue many operations after a STOP event, as in eventual recovery—perhaps too many. Yet, the benefits of even small b for $b > 0$ are not clear; in the hypothetical accounting firm of Section 3.1, bounded recovery ensures only that data is lost a bounded number of times, which is b times too many. Thus, bounded recovery may be too restrictive but also not restrictive enough in practice.

The most restrictive variant of recovery is immediate recovery:

DEFINITION 3.3.4. A distributed protocol *immediately recovers* from faulty clients if after a STOP event is issued for a client the client issues no more operations.

Immediate recovery is useful in practice because it allows for the standard access control semantics—once a client’s access has been revoked, the client cannot issue any more operations. Revoking the access rights of a client can be modeled as issuing a STOP event for that client. Unfortunately, immediate recovery may restrict the liveness properties of a protocol.

THEOREM 3.3.5. Suppose a protocol using n processes implements an object in an asynchronous environment that tolerates at least one crash-fault with operations read, write, and revoke. Read returns the current value, write updates the value, and revoke removes write permission for a specific process. Then this protocol is not wait-free [49].

Proof. Let each process begin with write access. Each process will attempt to write its proposed value to the object, revoke write access to every other process, and then read the current value of the object. Because reading comes after revoking, and because immediate recovery ensures that no writes complete after revocation, this algorithm implements n-process consensus, where the value read is the value agreed upon. Thus, by the FLP impossibility result [40], the protocol cannot always be live. □

Though Theorem 3.3.5 precludes distributed objects with wait-free read, write, and revoke operations that tolerate even a single crash, Section 3.4 will describe a protocol that offers immediate recovery with wait-free read and write operations, but a revoke operation that is not wait-free. In other words, a storage object can provide wait-free read and write operations, with the liveness penalty of immediate recovery only experienced when a revoke operation makes the penalty unavoidable.

3.4 A Wait-free Storage Protocol with Immediate Recovery

This section presents a protocol that offers immediate recovery with wait-free read and write operations (revoke operations are not wait-free). The protocol reads and writes from a single block. An array of blocks can be formed by running multiple instances of the protocol in parallel. The protocol requires $3f+1$ total servers to tolerate f Byzantine faulty servers, and it tolerates any number of Byzantine faulty clients. The network is assumed to be asynchronous. The client retries sending requests across the network until it receives enough responses to continue, but this retry pattern is not shown. The client has access to a replicated state machine subprotocol, such as PBFT or Zyzzyva [24, 55] which will be used for the revoke operation.

The protocol is comprised of three operations: Write, Read, and Revoke. The protocol is based on the timestamp paradigm found in many prior protocols (e.g., [44, 48, 66]), and is similar to that of Liskov and Rodrigues [66]. For a write, a client chooses a timestamp higher than the most recently completed write, and is said to write its block at that timestamp. A reader finds the block with the highest timestamp. If two blocks share the same timestamp, the tie is broken deterministically. In particular, in this protocol the block value that would represent the greater binary number is chosen. If the two block values are the same, either block is returned.

The protocol is designed for simplicity, as to prove that storage protocols can offer immediate recovery and wait-free read and write operations. The protocol was not designed for efficiency, and there are a number of optimizations that are not considered. Because the protocol is so similar to prior protocols (e.g.,the protocol of Liskov and Rodrigues [66]), the protocol presentation is kept as informal as possible.

3.4.1 Write

The write subprotocol is comprised of three rounds: First, the client finds the highest ts. Second, the client sends a prepare request, which includes the block value. Third, the client sends a commit request, which includes proof that prepare requests were received by $2f+1$ servers. In the first round, the client requests the highest ts value from each server. Each correct server returns with the greatest ts of any commit request processed. After receiving $2f+1$ responses, the client chooses a ts one greater than the greatest ts value returned.

The client then sends a signed prepare request to all servers, which consists of the chosen ts and the block. Each correct server stores the signed prepare request if it has the greatest $\langle \mathsf{ts}, \mathsf{block} \rangle$ pair from this client so far, and returns with a signed response which consists of a tag denoting this is a prepare response, the ts, and the block (in practice, only a collision-resistant hash of the block is required). If a client's write privileges have been revoked, a server may return a FAILURE message, which includes the signed revocation request, in which case the write fails, the write protocol returns FAILURE to the application, and a correct client no longer issues write operations.

After receiving $2f+1$ prepare responses, the client forms a *prepare certificate*, which is just the pair $\langle \mathsf{ts}, \mathsf{block} \rangle$ and the $2f+1$ signed prepare responses. The client sends a commit request to all servers, which consists of the prepare certificate. Each correct server stores the prepare certificate as the new most recent write if the pair $\langle \mathsf{ts}, \mathsf{block} \rangle$ is the greatest received from any client so far, even if the client has had its write privileges revoked. A correct server then sends an acknowledgment to the client. Upon receiving $2f+1$ acknowledgments of the commit request, the write is complete, and the write protocol returns SUCCESS to the application.

Liveness: The write subprotocol is wait-free. Each round consists of the client sending a request to $3f+1$ servers and waiting for responses from $2f+1$ servers. Eventually, the $2f+1$ correct servers will return some response, and the client will continue.

3.4.2 Read

The read subprotocol is comprised of two rounds. First, the client asks each server for the prepare certificate provided by a commit request with the highest timestamp. (Or some default block if no block has been committed.) After receiving $2f+1$ prepare certificates, the client chooses the prepare certificate with the highest timestamp. Second, the client writes back a commit request consisting of the prepare certificate to each server and waits for $2f+1$ acknowledgments. The read subprotocol then returns the block from that prepare certificate to the application.

Liveness: The read subprotocol is wait-free for the same reasons that the write subprotocol is wait-free.

3.4.3 Revoke

The revoke subprotocol has three rounds, in which a client called the *revoker* revokes write privileges for another client. First, the revoker asks each server for the prepare request with the greatest $\langle \mathsf{ts}, \mathsf{block} \rangle$ pair for the client being revoked in a signed revocation request (or a default block). Each server promises to not accept future prepare requests from this client (though a client can still complete a commit request if it already has a prepare certificate), and each server stores the signed revocation request. The revoker gathers $2f+1$ such prepare requests and such promises not to allow further prepares.

Second, the revoker sends these $2f+1$ non-matching prepare requests to the replicated state machine subprotocol. If this is the first revocation request for the client being revoked, the replicated state machine protocol returns the prepare request with the greatest $\langle \mathsf{ts}, \mathsf{block} \rangle$ pair, along with a signed message that allows the prepare request to be used as a prepare certificate. If this is not the first revocation request for the client being revoked, the replicated state machine protocol returns the prepare request it sent for the first revocation.

Third, the revoker uses the signed message and the prepare request to write back a commit request to each server. After $2f+1$ acknowledgments, the revoker is done and returns SUCCESS to the application.

An agreement subprotocol, as implied by Theorem 3.3.5, is provided in the second round by the replicated state machine. The agreement subprotocol ensures that a unique prepare request is the final prepare request committed during write back. Otherwise, two concurrent revoke operations could write back different values, which would allow a write for the client being revoked to appear after the first revoke operation completes.

Liveness: The revoke subprotocol inherits the liveness properties of the underlying replicated state machine subprotocol. The non-state-machine requests consist of the revoker sending a request to $3f+1$ servers and waiting for $2f+1$ responses, as in the write and read subprotocols, so the revoker will continue once the $2f+1$ correct servers respond.

3.4.4 Linearizability and Immediate Recovery

To see why this protocol is linearizable from the perspective of correct clients, consider the sequence of read and write operations from correct clients ordered by $\langle \mathsf{ts}, \mathsf{block} \rangle$ pairs, where operations with the same $\langle \mathsf{ts}, \mathsf{block} \rangle$ value are ordered by real-time precedence, with writes preceding reads whenever possible. Insert revoke operations by correct revoking clients as late as possible while respecting real-time precedence, but before any subsequent FAILURE for a write by a correct client being revoked. Finally, if the value of block for the write that immediately precedes a read does not match the value of block read, insert a write by a faulty client that is not yet revoked for the $\langle \mathsf{ts}, \mathsf{block} \rangle$ pair that is read. Insert the write immediately after the preceding read or write for a different $\langle \mathsf{ts}, \mathsf{block} \rangle$ pair, but before any revoke for the faulty client. Call this sequence of operations sequential history S.

It is always possible to insert a write for a faulty client as described above. Suppose a correct client reads a $\langle \mathsf{ts}, \mathsf{block} \rangle$ pair, but that the read response precedes the invocation any write by a correct client for that pair. The read returns a signed prepare certificate, which means that a signed prepare request must have been generated for some client for the pair $\langle \mathsf{ts}, \mathsf{block} \rangle$. If the prepare request was not signed by a correct client, it must correspond to a faulty client.

Note that if a read or write precedes another read or write, the $\langle \mathsf{ts}, \mathsf{block} \rangle$ pair will not decrease because write back ensures that the $\langle \mathsf{ts}, \mathsf{block} \rangle$ pair committed to at least $2f+1$ servers is at least as great as that of the first read or write. Thus, the subsequence of S consisting of each event for any correct client is the same as the the subsequence of events for that correct client in history H, the sequence of events from all correct clients. Each read, write, and revoke is present, and their relative ordering is consistent with real-time precedence.

By construction, the write that immediately precedes a read writes a matching value of block. Furthermore, if a revoke for a correct client precedes a write by that client, one of the $2f+1$ servers that promised to reject future prepare requests will return FAILURE, so the write will return failure. Also, if a write by a correct client returns failure, then the client received a revocation request that must have been signed by a client prior to the write invocation. Thus, S is a legal sequential history. Also, each response that precedes some invocation in H precedes that invocation in S. Thus, the protocol satisfies the natural Byzantine extension to linearizability.

Immediate recovery: Immediate recovery follows from Definition 3.3.4, the definition of the revoke operation in this protocol, and because the read, write, and revoke protocol described in this section satisfies the natural Byzantine extension to linearizability.

3.5 Conclusion

Distributed protocols use correctness arguments to ensure that applications experience the expected operational semantics. Unfortunately, correctness arguments are challenging in the presence of faulty clients because traditional notions of correctness, such as linearizability, assume that all nodes are correct. This chapter presented the natural extension to linearizability in environments with faulty clients. It also proposed *immediate recovery*, a new correctness criterion that ensures that faulty clients cannot affect the state of a storage system after being removed. Furthermore, the chapter demonstrated that a Byzantine fault-tolerant storage protocol can provide wait-free reads and writes yet still ensure immediate recovery.

Chapter 4

Low-Overhead Byzantine Fault-Tolerant Storage

Distributed storage systems must tolerate faults other than crashes as such systems grow in size and importance. Protocols that can tolerate arbitrarily faulty behavior by components of the system are said to be Byzantine fault-tolerant [65]. Most Byzantine fault-tolerant protocols are used to implement replicated state machines, in which each request is sent to a server replica and each non-faulty replica sends a response. Replication does not introduce unreasonable overhead when requests and responses are small relative to the processing involved, but for distributed storage, large blocks of data are often transferred as a part of an otherwise simple read or write request. Though a single server can return the block for a read request, the block must be sent to each server for a write request.

A storage protocol can reduce the amount of data that must be sent to each server by using an *m*-of-*n* erasure code [84, 85]. Each block is encoded into *n* fragments such that any *m* fragments can be used to decode the block. Unfortunately, existing protocols that use erasure codes struggle with tolerating Byzantine faulty clients. Such clients can write inconsistently encoded fragments, such that different subsets of fragments decode into different blocks of data.

Existing protocols that use erasure codes either provide each server with the entire block of data or introduce expensive techniques to ensure that fragments decode into a unique block. In the first approach, the block is erasure-coded at the server [18], which saves disk bandwidth and capacity but not network bandwidth. The second approach [44] saves network bandwidth but requires additional servers and a relatively expensive verification procedure for read operations. Furthermore, in this approach, all writes must be versioned because clients may need to read several versions of a block before read verification succeeds, and a separate garbage collection protocol must be run to free old versions of a block [3].

This chapter takes a different approach. It proposes novel mechanisms to optimize for the common case when faults and concurrency are rare. These optimizations minimize the number of rounds of communication, the amount of computation, and the number of servers that must be available at any time. Also, this chapter employs the homomorphic fingerprinting primitive developed in Chapter 2, to ensure that a block is encoded correctly. Homomorphic fingerprinting eliminates the need for versioned storage and a separate garbage collection protocol, and it minimizes the verification performed during read operations.

Analysis and measurements of a distributed block storage prototype demonstrate throughput close to that of a system that tolerates only benign crash faults and well beyond the throughput realized by competing Byzantine fault-tolerant approaches. The protocol achieves within 10% of the throughput of an ideal crash fault-tolerant system when reading or writing 64 kB blocks of data and tolerating up to 6 faulty servers and any number of faulty clients. Across a range of values for the number of faulty servers tolerated, the protocol outperforms competing approaches during write or read operations by more than a factor of two.

4.1 Background

Reliability has long been a primary requirement of storage systems. Thus, most non-personal storage servers (whether disk arrays or file servers) are designed to tolerate faults of at least some components. Until recently, tolerance of any single component fault was considered sufficient by many, but larger systems have pushed developers toward tolerating multiple component faults [31, 41].

Faults are tolerated via redundancy. In the case of storage systems, data is stored redundantly across multiple disk drives in order to tolerate faults in a subset of them. Two common forms of data redundancy are replication and m-of-n erasure coding, in which a block of data is encoded into n fragments such that any m can be used to reconstruct the original block. For disk arrays, the trade-off between the two has been extensively explored for two-way mirroring (i.e., two replicas) versus RAID-5 (i.e., $(n-1)$-of-n erasure coding) or RAID-6 ($(n-2)$-of-n erasure coding). Mirroring performs well when the number of faults tolerated is small or when writes are small, RAID-5 and RAID-6 perform better for large writes, and all three perform well for reads [27, 106]. The Google File System (GFS) [41], for example, uses replication for a mostly-read workload and by default tolerates two faults.

For distributed storage, as the number of faults tolerated grows beyond two or three, erasure coding provides much better write bandwidth [100, 105]. A few distributed storage systems support erasure coding. For example, Zebra [47], xFS [8], and PanFS [77] support parity-based protection of data striped across multiple servers. FAB [91], Ursa Minor [1], and RepStore [108] support more general m-of-n erasure coding.

4.1.1 Beyond Crash Faults

A common assumption is that tolerance of crash faults is sufficient for distributed storage systems, but an examination of a modern centralized storage server shows this assumption to be invalid. Most such servers integrate various checksum and scrubbing mechanisms to detect non-crash faults. One common approach is to store a checksum with every block of data and then verify that checksum upon every read (e.g., ZFS [14] and GFS [41]). This checksum can be used to detect problems such as when the device driver silently corrupts data [14, 41] or when the disk drive writes data to the wrong physical location [1, 14].

For example, if a disk drive overwrites the wrong physical block, a checksum may be able to detect this corruption, but only if the checksum is not overwritten as well. To prevent the checksum from being incorrect along with the data, the checksum is often stored separately (e.g., with the metadata [14, 41]).

In short, various mechanisms are applied to detect and recover from non-crash faults in modern storage systems. These mechanisms are chosen and combined in an ad hoc manner based on the collective experience of the organizations that design storage systems. Such mechanisms are inherently ad hoc because no fault model can describe just the types of faults that must be handled in distributed storage; as systems change, the types of faults change. In the absence of a more specific fault model, general Byzantine fault tolerance [65] can be used to cover all possibilities.

4.1.2 The Cost of Byzantine Fault Tolerance

Byzantine fault-tolerant m-of-n erasure-coded storage protocols require at least $m+2f$ servers to tolerate f faulty servers. Requiring this many servers is less imposing than it sounds; modern non-Byzantine fault-tolerant erasure-coded storage arrays already use a similar number of disk drives. A typical storage array will have several primary disk drives that store unencoded data (e.g., $m = 5$) and a few parity drives for redundancy (e.g., $f = 2$). Beyond these drives, however, an array will often include a pool of hot spares with f or more drives (sometimes shared with neighboring arrays). The reason for this setup is that once a drive fails or becomes otherwise unresponsive, the storage array must either halt or decrease the number of faults that it tolerates unless it replaces that drive with a hot spare. This setup is similar to providing $m+f$ responsive drives to a Byzantine fault-tolerant protocol but making an additional f drives available as needed (for a similar approach, see Rodrigues et al. [89]). In other words, the specter of additional hardware should not scare developers away from Byzantine fault tolerance.

Byzantine fault-tolerant protocols often require additional computational overhead. For example, the protocol proposed in this chapter requires data to be cryptographically hashed for each write and read operation. This overhead, however, is less significant than it appears for two reasons. First, data must be hashed anyway if it is to be authenticated when sent over the network. Of course, data must be structured properly to use a hash for both authentication and fault tolerance. Second, many modern file systems hash data anyway. For example, ZFS supports hashing all data with SHA-256 [99], and EMC's Centera hashes all data with either MD5 or a concatenation of MD5 and SHA-256 to provide content addressed storage [82].

4.1.3 Byzantine Fault-Tolerant Storage

Many Byzantine fault-tolerant protocols are used to implement replicated state machines. Implementations of recent protocols can be quite efficient due to several optimizations. For example, Castro and Liskov eliminate public-key signatures from the common case by replacing signatures with message authentication codes (MACs) and lazily retrieving signatures in cases of failure [23]. Abd-El-Malek et al. use aggressive optimistic techniques and quorums to scale as the number of faults tolerated is increased, but their protocol requires $5f+1$ servers to tolerate f faulty servers [2]. Cowling et al. use a hybrid of these two protocols to achieve good performance with only $3f+1$ servers [32]. Kotla and Dahlin further improve performance by using application-specific information to allow parallelism [57]. Though a Byzantine fault-tolerant replicated state machine protocol can be used to implement a block storage protocol, doing so requires writing data to at least $f+1$ replicas.

When writing large blocks of data and tolerating multiple faults, a Byzantine fault-tolerant storage protocol should provide erasure-coded fragments to each server to minimize the bandwidth overhead of redundancy. Writing erasure-coded fragments has been difficult to achieve because servers must ensure that a block is encoded correctly without seeing the entire block. Goodson et al. introduced PASIS, a Byzantine fault-tolerant erasure-coded block storage protocol [44]. In PASIS, servers do not verify that a block is correctly encoded during write operations. Instead, clients verify during read operations that a block is correctly encoded.

This technique avoids the problem of verifying erasure-coded fragments but introduces a few new ones. First, fragments must be kept in versioned storage [97] because clients may need to read several versions of a block before finding a version that is encoded correctly. Second, the read verification process is computationally expensive. Third, PASIS requires $4f+1$ servers to tolerate f faults. Fourth, a separate garbage collection protocol must eventually be run to free old versions of a block. A lazy verification protocol, which also performs garbage collection, was proposed to reduce the impact of read verification by performing it in the background [3], but this protocol consumes significant bandwidth.

Cachin and Tessaro introduced AVID [18], an asynchronous verifiable information dispersal protocol, which they used to build a Byzantine fault-tolerant block storage protocol that requires only $3f+1$ servers to tolerate f faults [19]. In AVID, a client sends each server an erasure-coded fragment. Each server sends its fragment to all other servers such that each server can verify that the block is encoded correctly. This all-to-all communication, however, consumes slightly more bandwidth in the common case than a typical replication protocol. Chapter 2 provides a protocol that reduces this overhead but still requires all-to-all communication and the encoding and hashing of $3f+1$ fragments (Section 2.3). These shortcomings are addressed in Section 4.2.1.

Many of the problems in PASIS are caused by the need to handle Byzantine faulty clients. Faulty clients should be tolerated in a Byzantine fault-tolerant storage system to prevent such clients from forcing the system into an inconsistent state. For example, though a faulty client can corrupt blocks for which it has write permissions, it must not be allowed to write a value that is read as two different blocks by two different correct clients; if not, a faulty client at a bank, e.g., could provide one account balance to the auditors but another to the ATM. Liskov and Rodrigues [66] propose that servers provide public-key signatures to vouch for the state of the system. This technique can be used to tolerate Byzantine faulty clients in a quorum system. In the next section, this technique is adapted as to use only MACs and pseudo-random nonce values in a PASIS-like protocol.

4.2 The FP Protocol

This section describes the block storage protocol, which is called the FP protocol to reflect its usage of homomorphic fingerprinting. A separate instance is executed for each block, so this section does not discuss block numbers. Section 4.2.1 describes the design of the protocol, including how it builds on prior protocols. Section 4.2.2 describes the system model. Section 4.2.3 provides pseudo-code for write and read operations. Section 4.2.4 discusses liveness and linearizability.

4.2.1 Design

Consider the following replication-based protocol [66], which requires $3f + 1$ servers to tolerate f faults. To write a block, a client hashes the block and sends the hash to all servers (the prepare phase). The servers respond with a signed message containing the hash and a logical timestamp, which is always greater than any timestamp that the server has seen. If there are not at least $2f + 1$ matching timestamps, the client requests that each server sign a new message using the greatest timestamp found. The client commits the write by sending the entire block along with $2f + 1$ signed messages with matching timestamps to each server. The server verifies that the signatures are valid and that the hash of the block matches the hash in the signed message. Because this protocol uses public-key signatures, the client can verify the responses from the prepare phase before it attempts to commit the write.

To read the block, the client queries all servers for their most recent timestamp and the $2f + 1$ signatures generated in the prepare phase. The client reads the block with the greatest timestamp and $2f + 1$ correctly signed messages. The signatures allow the the client to verify that $2f + 1$ servers provided signatures in the prepare phase, which ensures that some client invoked a write of this block at this timestamp (at least one of these servers is correct and, hence, would only provide signatures to a client in the prepare phase). To ensure that other clients see this block, the client writes it back to any servers with older timestamps.

Sending the entire block causes overhead that could be eliminated by sending erasure-coded fragments instead. Writing erasure-coded fragments, however, poses a problem, in that servers can no longer agree on what is being written. A faulty or malicious client can write fragments that decode to different blocks depending upon which subset of fragments is decoded. PASIS [44] uses a cross-checksum [43], which is a set of hashes of the erasure-coded fragments, to detect such inconsistencies. To write a block, a client requests the most recent logical timestamp from all servers in the first round. In the second round, the client sends each server its fragment, the cross-checksum, and the greatest timestamp found. Unfortunately, PASIS requires $4f + 1$ servers to tolerate f faults, so $4f + 1$ fragments must be encoded and hashed, which is a significant expense. To read a block, the client reads fragments and cross-checksums from each server, starting with the most recent timestamp, until it finds m fragments whose hashes correspond to their respective locations in the cross-checksum. From these fragments, the client decodes the block, re-encodes n fragments, and recomputes the cross-checksum. If this cross-checksum does not match the one provided by the servers, then the write operation for that timestamp was invalid and the client must try reading fragments at an earlier timestamp. If cross-checksums match, the client writes fragments back to servers as needed.

Improvements

The replication-based protocol requires only $3f + 1$ servers but relies on public-key signatures and replication. PASIS improves write bandwidth but has a number of drawbacks, as discussed in Section 4.1.3. The protocol proposed in this paper improves on these approaches with the following four techniques.

No public-key cryptography: As in the replication-based protocol [66], the response to a prepare request in the protocol includes an authenticated timestamp and checksum that allows the client to progress to the commit phase once enough timestamps match. Castro and Liskov [23] avoid using signatures for authentication by using message authentication codes (MACs) in the common case but lazily retrieving signatures when needed. This lazy retrieval technique does not work for the Liskov and Rodrigues protocol, however, because signatures are stored for later use to vouch for the state of the system [66, Section 3.3.2].

Instead, signatures are avoided altogether and rely entirely on MACs and random nonce values. All servers share pairwise MAC keys (clients do not create or verify MACs). Each server provides MACs to the client in the prepare phase, which the client sends in the commit phase to prove that enough servers successfully completed prepare requests at a given timestamp. Of course, a faulty server may provide faulty MACs during the prepare phase, or it may reject valid MACs during the commit phase. To recover from this, a client may need to gather more MACs from the prepare phase after it has entered the commit phase, but a commit eventually completes.

The replication-based protocol also uses signatures to allow servers to prove to a reader that a block was written by some client; that is, to prevent a server from returning fabricated data. Instead of signatures, each server provides a pseudo-random nonce value in the prepare phase of the protocol. The client aggregates these values and provides them to each server in the commit phase. During a read operation, a server provides the client with these nonce values to prove that some client invoked a write at a specific timestamp, as will be described in Section 4.2.3 and Lemma 4.2.5 of Section 4.2.4.

Early write: Before committing a write, a correct server must ensure that enough other correct servers have fragments for this write, such that a reader will be able to reconstruct the block. The replication-based protocol does not face this problem because each server stores the entire block. The protocol proposed in this paper could solve this problem with another round of communication for servers to confirm receipt of a fragment. Instead, in the protocol and unlike previous protocols, clients send erasure-coded fragments in the first round of the prepare phase, which saves a round of communication. With this approach, a faulty client may send a fragment in the first phase without committing the write. As in PASIS [3], a server may limit this by rejecting a write from a client with too many uncommitted writes.

Partial encoding: PASIS encodes and hashes fragments for all n servers. Encoding this many fragments is wasteful because f servers may not be involved in a write operation. Instead, the protocol encodes and hashes fragments for only the first $n-f$ servers, which lowers the computational overhead. This technique is called partial encoding because the block is only partially encoded for most write operations. The computational savings are significant: for $m = f+1$, the protocol encodes only $2f+1$ fragments, which is the same number encoded by a non-Byzantine fault-tolerant erasure-coded protocol. Many m-of-n erasure codes encode the first m fragments by dividing a block into m fragments (such codes are said to be systematic), which takes little if any computation. Hence, encoding $2f+1$ fragments requires computing f values, whereas encoding $4f+1$ fragments (as in PASIS) requires computing $3f$ values.

The drawback of this approach is that if one of the first $m+f$ servers is non-responsive or faulty, the client may need to send the entire block to convince another server that its fragment corresponds to the checksum. This procedure is expensive: not only does it consume extra bandwidth, but the server must verify the block against the checksum. To verify a block, the server encodes the block into $m+f$ fragments, hashes each fragment, and compares these hashes to the checksum provided by the client. If the hashes match the checksum, the server encodes its fragment from the block. Fortunately, the first $m+f$ servers should rarely be non-responsive or faulty.

Distributed verification of erasure-coded data: One problem in PASIS is that each server knows only the cross-checksum and its fragment, and so it is difficult for a server to verify that its fragment together with the corresponding fragments held by other servers form a valid erasure coding of a unique block. Chapter 2 solves this problem using a fingerprinted cross-checksum. The fingerprinted cross-checksum includes a cross-checksum, as used in PASIS, along with a set of homomorphic fingerprints of the first m fragments of the block. The fingerprints are homomorphic in that the fingerprint of the erasure coding of a set of fragments is equal to the erasure coding of the fingerprints of those fragments. The overhead of computing homomorphic fingerprints is small compared to the cryptographic hashing for the cross-checksum.

The i^{th} fragment is said to be consistent with a fingerprinted cross-checksum if its hash matches the i^{th} index in the cross-checksum and its fingerprint matches the i^{th} erasure coding of the homomorphic fingerprints. Thus, a server can determine if a fragment is consistent with a fingerprinted cross-checksum without access to any other fragments. Furthermore, any two blocks decoded from any two sets of m fragments that are consistent with the fingerprinted cross-checksum are identical with all but negligible probability (Theorem 2.2.4). A server can check that a fragment is consistent with a fingerprinted cross-checksum shared by other servers on commit, allowing it to overwrite old fragments. Thus, only fragments that are in the process of being written must be versioned, obviating the need for on-disk versioning. This technique also eliminates most of the computational expense of validating the cross-checksum during a read operation.

Protocol Overview

This section provides an overview of the protocol. The protocol provides wait-free [49] writes and obstruction-free [50] reads of constant-sized blocks while tolerating a fixed number of Byzantine servers and an arbitrary number of Byzantine clients in an asynchronous environment. Figure 4.2.1 provides an outline of the pseudo-code for both write operations and read operations. The line numbers in Figure 4.2.1 match those of Figures 4.2.2 and 4.2.3. Figure 4.2.2 provides detailed pseudo-code for a write operation and is described line-by-line in Section 4.2.3. Figure 4.2.3 provides detailed pseudo-code for a read operation and is described line-by-line in Section 4.2.3.

To write a block, a client encodes the block into $m+f$ fragments, computes the fingerprinted cross-checksum, and sends each server its fragment and the fingerprinted cross-checksum (lines 1000–1106). The server responds with a logical timestamp, a nonce, and a MAC for each server of the timestamp, fingerprinted cross-checksum, and nonce (line 1405). If timestamps do not match, the client requests new MACs at the greatest timestamp found (line 1114). Unlike the signatures in the Liskov and Rodrigues protocol, the client cannot tell if these MACs are valid.

The client then commits this write by sending the timestamp, fingerprinted cross-checksum, nonces, and MACs to each server (line 1128). A correct server may reject a commit with MACs from faulty servers or a faulty server may reject a commit with MACs from correct servers. Faults should be uncommon, but when they occur, the client must contact another server. The client can either try the commit at another server or it can send the entire block to another server in order to garner another prepare response. A write operation returns after at most three rounds of communication with correct servers. Because faults and concurrency are rare, the timestamps received in the first

```
c_write(B):
1000:  $d_1, \ldots, d_{m+f} \leftarrow \mathsf{encode}_{1,\ldots,m+f}(B)$            /* Partial encoding */
1001:  for ($i \in \{1,\ldots,m+f\}$) do $\mathsf{fpcc.cc}[i] \leftarrow \mathsf{hash}(d_i)$
1002:  for ($i \in \{1,\ldots,m\}$) do $\mathsf{fpcc.fp}[i] \leftarrow \mathsf{fingerprint}(\mathsf{hash}(\mathsf{fpcc.cc}), d_i)$
...:   for ($i \in \{1,\ldots,m+f\}$) do
1106:    $Prepare[i] \leftarrow S_i$.s_rpc_prepare_frag($d_i$, ts, fpcc)
1405:    /* Server returns $\langle \mathsf{ts}, \mathsf{nonce}, \langle \mathsf{MAC}_{i,j}(\langle \mathsf{ts}, \mathsf{fpcc}, \mathsf{nonce}\rangle)\rangle_{1 \le j \le n}\rangle$ */
1114:    if ($Prepare[i].\mathsf{ts} \neq \mathsf{ts}$) then                    /* Retry prepare */
...:       ...                                            /* See lines 1109-1117 */
...:   for ($i \in \{1,\ldots,m+f\}$) do
1128:    $S_i$.s_rpc_commit(ts, fpcc, $Prepare$)

c_read():
...:   for ($i \in \{1,\ldots,2f+1\}$) do
1605:    $\langle \mathsf{ts}, \mathsf{fpcc}\rangle \leftarrow S_i$.s_rpc_find_timestamp()
...:   for ($i \in \{1,\ldots,m\}$) do
1620:    $d_i \leftarrow S_i$.s_rpc_read(ts, fpcc)
1817:  $B \leftarrow \mathsf{decode}(d_1, \ldots, d_m)$
...:   /* Client verifies the consistency of block B in c_find_block */
1626:  return B
```

Figure 4.2.1: Pseudo-code outline. Line numbers match Figures 4.2.2 and 4.2.3.

round of prepare will often match, which allows most write operations to complete in only two rounds.

To read a block, a client requests timestamps and fingerprinted cross-checksums from $2f+1$ servers (line 1605) and fragments from the first m servers (line 1620). If the m fragments are consistent with the most recent fingerprinted cross-checksum, and if the client can determine that some client invoked a write at this timestamp (using nonces as described in Section 4.2.3), a block is decoded (line 1817) and returned (line 1626). (In Figure 4.2.3, the client verifies read responses and decodes a block in **c_find_block**.) Most read operations return after one round of communication with correct servers. If a concurrent write causes a fragment to be overwritten, however, the client may be redirected to a later version of the block, as described in Section 4.2.3.

4.2.2 System Model

The point-to-point communication channel between each client and server is authenticated and in-order, which can be achieved in practice with little overhead. Communication channels are reliable but asynchronous, i.e., each message sent is eventually received, but there is no bound assumed on message transmission delays. Reliability is assumed for presentational convenience only; the protocol can be adapted to unreliable channels as discussed by Martin et al. [73, Section 4.3].

Up to f servers and an arbitrary number of clients are Byzantine faulty, behaving in an arbitrary manner. An adversary can coordinate all faulty servers and clients. To bound the amount of storage used to stage fragments for in-progress writes, there is a fixed upper bound on the number of clients in the system and on the number of prepare requests from each client that are not followed by subsequent commits.

It is assumed that there is a negligible probability that a MAC can be forged or that a hash collision or preimage can be found. The fingerprinted cross-checksum requires that the hash function acts as a random oracle [12]. All servers share pairwise MAC keys. The value labeled nonce must not be disclosed to parties except as prescribed by the protocol, in order to prove Lemma 4.2.5 in Section 4.2.4. This value is small and can be encrypted at little cost.

The protocol tolerates f Byzantine faulty servers and any number of faulty clients given an m-of-n erasure code and $n = m+2f$ servers, where $m \geq f+1$. As in PASIS, the protocol may be deployed with $m > f+1$ to achieve higher bandwidth for fixed f.

4.2.3 Detailed Pseudo-code

This section provides detailed pseudo-code for write and read operations. Pseudo-code for a write operation is described line-by-line in Section 4.2.3. Pseudo-code for a read operation is described line-by-line in Section 4.2.3. Presentation simplicity of the pseudo-code is chosen over optimizations that may be found in an actual implementation.

Notation

The protocol relies on concurrent requests that are described in the pseudo-code by remote procedure calls and coroutines. The **cobegin** and parallel bars represent the forking of parallel threads of execution. Such threads stop at **end cobegin**. The main thread continues to execute after forking threads; that is, the main thread does not wait to join forked threads at **end cobegin**. Threads are not preempted until they invoke a remote procedure call or wait on a semaphore. Semaphores are binary and default to zero. A WAIT operation waits on a semaphore, and SIGNAL releases all waiting threads. A return statement halts all threads and returns a value.

Each operation is assigned a logical timestamp, represented by the pair $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$. Timestamps are ordered according to the value of the integer ts or, if they share the same ts, by comparison of the binary value fpcc. Timestamps that share the same ts and fpcc are equal. The most recent commit at a server is represented as latest_commit; this value is initialized to $\langle 0, \text{NULL} \rangle$ before the protocol starts. Each server stages fragments for concurrent writes and stores committed fragments; both staged and committed fragments are kept in the store table.

A block is represented as B and a fragment as d in the pseudo-code. A cross-checksum, abbreviated to cc in the pseudo-code, contains n hashes, $\mathsf{cc}[1], \ldots, \mathsf{cc}[n]$. The fingerprinted cross-checksum, abbreviated to fpcc, contains m fingerprints, $\mathsf{fpcc.fp}[1], \ldots, \mathsf{fpcc.fp}[m]$, and $m+f$ hashes, $\mathsf{fpcc.cc}[1], \ldots, \mathsf{fpcc.cc}[m+f]$.

The $\mathsf{encode}_j(\mathsf{B})$ function encodes block B into its j^{th} erasure-coded fragment. The $\mathsf{decode}(\ldots)$ function will decode the first m fragments provided in its arguments, and the index of each fragment is passed implicitly. Abbreviate $\mathsf{encode}_j(\mathsf{decode}(\mathsf{d}_{i_1}, \ldots, \mathsf{d}_{i_m}))$ to $\mathsf{encode}_j(\mathsf{d}_{i_1}, \ldots, \mathsf{d}_{i_m})$. The $\mathsf{fingerprint}(\mathsf{h}, \mathsf{d})$ function fingerprints fragment d given random value h. The random value in the protocol is provided by a hash of the cross-checksum, which is secure so long as the hash function acts as a random oracle [12]. The homomorphism of the fingerprints provides the following property (Theorem 2.1.11): if $\mathsf{d}_1, \ldots, \mathsf{d}_n \leftarrow \mathsf{encode}_{1,\ldots,n}(\mathsf{B})$ and $\mathsf{fp}_i \leftarrow \mathsf{fingerprint}(\mathsf{h}, \mathsf{d}_i)$, then $\mathsf{fp}_{i_0} = \mathsf{encode}_{i_0}(\mathsf{fp}_{i_1}, \ldots, \mathsf{fp}_{i_m})$ for any set of indices $i_0, \ldots, i_m$.

```
c_write(B):                                   /* Wrapper function for c_dowrite */
1000: $d_1, \ldots, d_{m+f} \leftarrow \text{encode}_{1,\ldots,m+f}(B)$
1001: for $(i \in \{1,\ldots,m+f\})$ do fpcc.cc[$i$] $\leftarrow$ hash($d_i$)
1002: for $(i \in \{1,\ldots,m\})$ do fpcc.fp[$i$] $\leftarrow$ fingerprint(hash(fpcc.cc), $d_i$)
1003: c_dowrite(B, NULL, fpcc)

c_dowrite(B, ts, fpcc):                                          /* Do a write */
1100: Prepare[*] $\leftarrow$ NULL
1101: cobegin
1102: $||_{i\in\{1,\ldots,n\}}$                                /* Start worker threads */
1103:   /* Send fragment and fpcc to server, get MAC of fpcc and latest ts */
1104:   $d_i \leftarrow \text{encode}_i(B)$
1105:   if (fpcc.cc[$i$] = hash($d_i$)) then
1106:       Prepare[$i$] $\leftarrow$ $S_i$.s_rpc_prepare_frag($d_i$, ts, fpcc)
1107:   else Prepare[$i$] $\leftarrow$ $S_i$.s_rpc_prepare_block(B, ts, fpcc)   /* Bad fpcc.cc[$i$] */
1108:
1109:   /* Choose latest ts, if needed */
1110:   if (ts = NULL $\wedge$ $|\{j : Prepare[j] \neq \text{NULL}\}| = 2f+1$) then
1111:     ts $\leftarrow$ max{ts' : $\langle$ts', *$\rangle$ $\in$ Prepare}
1112:     SIGNAL(found_largest_ts)
1113:   if (ts = NULL) then WAIT(found_largest_ts)           /* Wait for chosen ts */
1114:   if (Prepare[$i$].ts $\neq$ ts) then           /* Need new prepare for chosen ts */
1115:     if (fpcc.cc[$i$] = hash($d_i$)) then
1116:         Prepare[$i$] $\leftarrow$ $S_i$.s_rpc_prepare_frag($d_i$, ts, fpcc)
1117:     else Prepare[$i$] $\leftarrow$ $S_i$.s_rpc_prepare_block(B, ts, fpcc)
1118:
1119:   /* Attempt commit */
1120:   if ($|\{j : Prepare[j].ts = ts\}| \geq m+f$) then SIGNAL(prepare_ready)
1121: end cobegin
1122:
1123: UnwrittenSet $\leftarrow \{1,\ldots,n\}$
1124: while (TRUE) do
1125:   WAIT(prepare_ready)
1126:   cobegin
1127:   $||_{i\in UnwrittenSet}$                                 /* Start worker threads */
1128:     if (SUCCESS = $S_i$.s_rpc_commit(ts, fpcc, Prepare)) then
1129:       UnwrittenSet $\leftarrow$ UnwrittenSet $\setminus \{i\}$
1130:     if ($|UnwrittenSet| \leq f$) then return SUCCESS
1131:   end cobegin

$S_i$.s_rpc_prepare_frag(d, ts, fpcc):        /* Grant permission to write fpcc at ts */
1200: fp $\leftarrow$ fingerprint(hash(fpcc.cc), d);   h $\leftarrow$ hash(d)
1201: fp' $\leftarrow$ encode$_i$(fpcc.fp[1], ..., fpcc.fp[$m$])
1202: if (fp = fp' $\wedge$ h = fpcc.cc[$i$]) then      /* Fragment is consistent with fpcc */
1203:   return $S_i$.s_prepare_common(ts, fpcc, d, NULL)
1204: else return FAILURE                                          /* Faulty client */

$S_i$.s_rpc_prepare_block(B, ts, fpcc):       /* Grant permission to write fpcc at ts */
1300: /* Called when partial encoding fails or when writing back fragments */
1301: cnt $\leftarrow$ 0
1302: for ($j \in \{1,\ldots,m+f\}$) do                              /* Validate block */
1303:   $d_j \leftarrow \text{encode}_j(B)$
1304:   fp $\leftarrow$ fingerprint(hash(fpcc.cc), $d_j$);   cc[$j$] $\leftarrow$ hash($d_j$)
1305:   fp' $\leftarrow$ encode$_j$(fpcc.fp[1], ..., fpcc.fp[$m$])
1306:   if (fp = fp' $\wedge$ cc[$j$] = fpcc.cc[$j$]) then cnt $\leftarrow$ cnt + 1
1307: if (cnt $\geq m$) then                  /* Found $m$ fragments consistent with fpcc */
1308:   for ($j \in \{m+f+1,\ldots,n\}$) do cc[$j$] $\leftarrow$ hash(encode$_j$(B))
1309:   return $S_i$.s_prepare_common(ts, fpcc, encode$_i$(B), cc)
1310: else return FAILURE                                          /* Faulty client */

$S_i$.s_prepare_common(ts, fpcc, d, cc):            /* Create the prepare response */
1400: if (ts = NULL) then ts $\leftarrow$ latest_commit.ts + 1
1401: nonce $\leftarrow$ MAC$_{i,i}$($\langle$ts, fpcc$\rangle$)
1402: if ($\langle$ts, fpcc$\rangle$ > latest_commit) then
1403:   nonce_hash $\leftarrow$ hash(nonce)
1404:   store[$\langle$ts, fpcc$\rangle$] $\leftarrow$ $\langle$d, cc, nonce_hash, NULL$\rangle$
1405: return $\langle$ts, nonce, $\langle$MAC$_{i,j}$($\langle$ts, fpcc, nonce$\rangle$)$\rangle_{1\leq j\leq n}$$\rangle$

$S_i$.s_rpc_commit(ts, fpcc, Prepare):                /* Commit write of fpcc at ts */
1500: if ($\langle$ts, fpcc$\rangle$ $\leq$ latest_commit) then return SUCCESS   /* Overwritten */
1501: Nonces $\leftarrow$ {$\langle j$, nonce$\rangle$ : Prepare[$j$] = $\langle$ts, nonce, $\langle$tag$_k\rangle_{1\leq k\leq n}$$\rangle$ $\wedge$
1502:                           tag$_i$ = MAC$_{j,i}$($\langle$ts, fpcc, nonce$\rangle$)}
1503: if ($|Nonces| \geq m+f$) then
1504:   $\langle$d, cc, nonce_hash, *$\rangle$ $\leftarrow$ store[$\langle$ts, fpcc$\rangle$]
1505:   store[$\langle$ts, fpcc$\rangle$] $\leftarrow$ $\langle$d, cc, nonce_hash, Nonces$\rangle$
1506:   for ($\langle$ts', fpcc'$\rangle$ < $\langle$ts, fpcc$\rangle$) do store[$\langle$ts', fpcc'$\rangle$] $\leftarrow$ NULL
1507:   latest_commit $\leftarrow$ $\langle$ts, fpcc$\rangle$
1508:   return SUCCESS
1509: else return FAILURE
```

Figure 4.2.2: Detailed write pseudo-code.

Write

Pseudo-code for write is provided in Figure 4.2.2. A write operation is invoked with a block of data as its argument and returns SUCCESS as its response. Write is divided into prepare and commit phases. The pseudo-code breaks write into a wrapper function, **c_write**, and a main function, **c_dowrite**, such that the main function can be reused for writing back fragments during a read operation. The wrapper function encodes the block into $m+f$ fragments and computes a fingerprinted cross-checksum (lines 1000–1002). It then calls **c_dowrite** (line 1003).

Prepare, lines 1100–1121: The prepare phase is described in the top half of **c_dowrite**. The client invokes **s_rpc_prepare_frag** at each of the first $m+f$ servers with its fragment and the fingerprinted cross-checksum (line 1106); ts will be NULL. A correct server S_i verifies that this fragment is consistent with the fingerprinted cross-checksum. To do so, it first computes the fingerprint and the hash of the fragment (line 1200). It then computes the i^{th} erasure coding of the homomorphic fingerprints in the fingerprinted cross-checksum (line 1201). Finally, it ensures that this erasure coding is equal to the fingerprint of this fragment, and that the hash is equal to the i^{th} hash in the cross-checksum (line 1202). If the fragment is consistent, the server prepares a response in **s_prepare_common** (line 1203).

Because of partial encoding, there are only $m+f$ erasure-coded fragments, so if one of the first $m+f$ servers is not responsive or if commit fails, the client may need to invoke **s_rpc_prepare_block** with the entire block (line 1107). A correct server S_i verifies that the erasure coding of the block contains at least m fragments that are consistent with the fingerprinted cross-checksum. To do so, it encodes the block into each of $m+f$ fragments (line 1303), computes the fingerprint and the hash of each fragment (line 1304), and computes the appropriate erasure coding of the homomorphic fingerprints in the fingerprinted cross-checksum (line 1305). It counts the number of fragments for which the fingerprint is equal to the erasure coding of the homomorphic fingerprints and the hash is equal to the appropriate hash in the fingerprinted cross-checksum (line 1306). If there are at least m such consistent fragments, server S_i computes the i^{th} fragment d_i and the rest of the cross-checksum of all n fragments and prepares a response in **s_prepare_common** (lines 1307–1309).

Invoking **s_rpc_prepare_block** allows a client to write a fragment that is not consistent with the fingerprinted cross-checksum, so long as this fragment can be erasure-coded from a block with m erasure-coded fragments that are consistent with the fingerprinted cross-checksum. This will be useful in the read protocol to ensure that any fragment can be written back as needed.

The response prepared in **s_prepare_common** consists of a ts, a nonce, and n MACs (one for each server). The ts may be provided by the client; if not, it is assigned to one greater than the ts portion of the logical timestamp used in the most recent commit (line 1400). The nonce is a pseudo-random value that is unique for each timestamp (line 1401); a MAC of the timestamp can be used to ensure this property. An array of n MACs is computed with the shared pairwise MAC keys (line 1405). The MACs are used in the commit phase to authenticate the timestamp and nonce, as well as to prove that enough correct servers stored consistent fragments.

If this timestamp is more recent than the most recently committed timestamp (line 1402), the nonce_hash, a preimage-resistant hash of the nonce, is computed (line 1403), and the fragment and nonce_hash are stored for future reads (line 1404). If **s_prepare_common** was called by **s_rpc_prepare_block**, the correct cross-checksum of all n fragments is also stored. (The NULL value on line 1404 is a placeholder that will be filled in **s_rpc_commit**.) The nonces and nonce_hashes are used to prove that a client invoked a write at this timestamp. Other protocols ensure this property in ways that would require more communication [18], public-key signatures [66], or $4f+1$ servers [44]; nonces are used here to avoid these mechanisms.

The client must wait for $2f+1$ responses before assigning a timestamp to this write. (The first $2f$ threads wait on line 1113 until a timestamp is assigned.) The timestamp is the pair $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$, where ts is the greatest ts value from the $2f+1$ responses. If a server provides a response with a different timestamp, the client must retry that request (lines 1114–1117).

Commit, lines 1123–1131: After $m+f$ servers have provided MACs in responses with matching timestamps, commit may be attempted (line 1120). The commit may fail if a faulty server provided one of these MACs or rejects a MAC from a correct server. But, eventually, at least $m+f$ correct servers will return responses with MACs that will be accepted by at least $m+f$ servers in commit. As prepare responses arrive, they are forwarded to all servers (line 1128). Thus, the threads from the prepare phase do not stop until all servers return responses or the commit phase completes. The commit phase completes and the client can return once $m+f$ servers (all but f) return SUCCESS (line 1130).

If a write has a lower timestamp than a previously committed write, a server can ignore it (line 1500). A correct server aggregates nonces from valid prepare responses (lines 1501–1502). A prepare response is valid if the MAC included for this server is a MAC of the timestamp and

```
c_read():                                                        /* Read a block */
1600: $Timestamp[*] \leftarrow$ NULL;    $State[*] \leftarrow \emptyset$
1601:
1602: /* Search for write timestamps */
1603: cobegin
1604: $||_{i \in \{1,\dots,3f+1\}}$                                  /* Start worker threads */
1605:     $Timestamp[i] \leftarrow$ S_i.s_rpc_find_timestamp()
1606:     SIGNAL(found_timestamp)
1607: end cobegin
1608:
1609: while (TRUE) do
1610:     WAIT(found_timestamp)
1611:     /* Try any timestamp greater or equal to $2f+1$ timestamps */
1612:     for ($\langle ts, fpcc \rangle : \langle ts, fpcc \rangle = Timestamp[i] \wedge \langle ts, fpcc \rangle \notin Tried \wedge$
1613:                 $|\{j : \langle ts, fpcc \rangle \geq Timestamp[j]\}| \geq 2f+1$) do
1614:         $Tried \leftarrow Tried \cup \{\langle ts, fpcc \rangle\}$
1615:         if ($ts = 0$) then return NULL                           /* No writes yet */
1616:
1617:         cobegin
1618:         $||_{i \in \{1,\dots,n\}}$                                /* Start worker threads */
1619:             $\langle \overline{ts}, \overline{fpcc} \rangle \leftarrow \langle ts, fpcc \rangle$          /* Thread local copy of variables */
1620:             $\langle data, gc\_redirect \rangle \leftarrow$ S_i.s_rpc_read($\overline{ts}, \overline{fpcc}$)
1621:             B $\leftarrow$ c_find_block($i, State, \overline{ts}, \overline{fpcc}$, data)
1622:
1623:             /* Write back fragments as needed and return the block */
1624:             if (B $\neq$ NULL) then
1625:                 c_dowrite(B, $\overline{ts}, \overline{fpcc}$)
1626:                 return B
1627:
1628:             /* Follow garbage collection redirection */
1629:             if (gc_redirect $\neq$ NULL $\wedge$ gc_redirect $> \langle \overline{ts}, \overline{fpcc} \rangle$) then
1630:                 $Timestamp[i] \leftarrow$ gc_redirect
1631:                 SIGNAL(found_timestamp)
1632:         end cobegin

S_i.s_rpc_find_timestamp():                            /* Return the latest commit */
1700: return latest_commit

c_find_block($i, State$, ts, fpcc, $\langle$d, cc, nonce_hash, $Nonces\rangle$):      /* Classify read */
1800: if (d $\neq$ NULL $\wedge$ cc $=$ NULL) then      /* Verify fragment-encoded arguments */
1801:     fp $\leftarrow$ fingerprint(hash(fpcc.cc), d)
1802:     fp' $\leftarrow$ encode_i(fpcc.fp[1], ..., fpcc.fp[$m$])
1803:     if (fp $\neq$ fp' $\vee$ hash(d) $\neq$ fpcc.cc[$i$]) then return NULL
1804:
1805: if (d $\neq$ NULL $\wedge$ cc $\neq$ NULL) then         /* Verify block-encoded arguments */
1806:     if (hash(d) $\neq$ cc[$i$]) then return NULL
1807:
1808: /* Update state and count preimages */
1809: $State[\langle ts, fpcc \rangle] \leftarrow State[\langle ts, fpcc \rangle] \cup \{\langle i, d, cc, nonce\_hash, Nonces \rangle\}$
1810: npreimages $\leftarrow |\{j : \langle j, *, *, nonce\_hash', * \rangle \in State[\langle ts, fpcc \rangle] \wedge$
1811:                        $\langle *, *, *, *, \{*, \langle j, nonce' \rangle, *\} \rangle \in State[\langle ts, fpcc \rangle] \wedge$
1812:                        $nonce\_hash' = hash(nonce')\}|$
1813: if (npreimages $< f+1$) then return NULL
1814:
1815: /* Try to decode */
1816: $Frags \leftarrow \{d' \neq NULL : \langle *, d', NULL, *, * \rangle \in State[\langle ts, fpcc \rangle]\}$
1817: if ($|Frags| \geq m$) then return decode($Frags$)
1818: else for ($cc' \neq NULL : \langle *, *, cc', *, * \rangle \in State[\langle ts, fpcc \rangle]\}$) do
1819:     $Frags' \leftarrow \{d' \neq NULL : \langle *, d', cc', *, * \rangle \in State[\langle ts, fpcc \rangle]\}$
1820:     if ($|Frags \cup Frags'| \geq m$) then
1821:         cnt $\leftarrow 0$;    B $\leftarrow$ decode($Frags \cup Frags'$)
1822:         for ($j \in \{1, \dots, m+f\}$) do                          /* Validate block */
1823:             $d_j \leftarrow encode_j(B)$
1824:             fp $\leftarrow$ fingerprint(hash(fpcc.cc), $d_j$);    h $\leftarrow$ hash($d_j$)
1825:             fp' $\leftarrow$ encode_j(fpcc.fp[1], ..., fpcc.fp[$m$])
1826:             if (fp $=$ fp' $\wedge$ h $=$ fpcc.cc[$j$]) then cnt $\leftarrow$ cnt $+ 1$
1827:         if (cnt $\geq m$) then             /* Found $m$ fragments consistent with fpcc */
1828:             return B
1829:
1830: return NULL                                                  /* No block found */

S_i.s_rpc_read(ts, fpcc):                          /* Read the fragment at $\langle ts, fpcc \rangle$ */
1900: if (store[$\langle ts, fpcc \rangle$] $=$ NULL $\wedge$ latest_commit $> \langle ts, fpcc \rangle$) then
1901:     return $\langle\langle$NULL, NULL, NULL, NULL$\rangle$, latest_commit$\rangle$
1902: else return $\langle$store[$\langle ts, fpcc \rangle$], NULL$\rangle$
```

Figure 4.2.3: Detailed read pseudo-code.

nonce computed with the proper pairwise key. If there are at least $m+f$ nonces from valid prepare requests, then at least m correct servers stored a fragment, so commit will succeed. The NULL value from line 1404 is filled in with these nonces (lines 1504–1505). This will become the new most recent write (line 1507). If a client tries to read a fragment with a lower timestamp, it can be redirected to this write, so earlier fragments can be garbage collected (line 1506). Hence, a server must stage fragments for concurrent writes but store only the most recently committed fragments.

Read

Pseudo-code for a read operation is provided in Figure 4.2.3. A read operation is invoked with no arguments and returns a block as its response. A read operation is divided into two phases, "find timestamps" and "read timestamp." The client searches for the timestamps of the most recently committed write at each server. As timestamps arrive, the client tries reading at any timestamp greater than or equal to $2f+1$ other timestamps.

Find timestamps, lines 1600–1615: The client queries each of the first $3f+1$ servers for the timestamp of its most recently committed write (lines 1602–1607). As timestamps arrive, the client tries reading at any timestamp that it has yet to try already and that is greater than or equal to $2f+1$ other timestamps (lines 1610–1614). If no writes have been committed, the value latest_commit (line 1700) defaults to $\langle 0, \text{NULL} \rangle$; if $2f+1$ or more servers return $\langle 0, \text{NULL} \rangle$, no writes have returned yet so a NULL block is returned (line 1615).

Read timestamp, lines 1617–1632: To read a fragment, the client invokes **s_rpc_read** at each server (line 1620). A correct server returns the fragment along with the other data stored during write (line 1902). The client processes each response from **s_rpc_read** with the helper function **c_find_block** (line 1621). This function verifies that the fragment is valid (lines 1800–1806), determines whether a client invoked this write (lines 1810–1813), and decodes a block if possible (lines 1815–1828). If a correct server has no record of this fragment but knows of a more recent write, it returns the timestamp of the more recent write (line 1901). The client follows such garbage collection redirections if a block is not found (lines 1628–1631).

If a correct server received a fragment in a successful call to **s_rpc_prepare_frag**, the value cc will be set to NULL and the client will verify that the fragment is consistent with the fingerprinted cross-checksum (lines 1801–1803). If a correct server received a fragment in a successful call to **s_rpc_prepare_block**, the value cc will be the cross-checksum of all n fragments. The client verifies that the cross-checksum cc matches at least this fragment (line 1806). If either verification fails, the response is ignored (lines 1803 and 1806). Otherwise, the client records this fragment in the state for this timestamp (line 1809) and tries to determine whether a client invoked this write (lines 1810–1813). If there are at least $f+1$ nonces that are the preimages of nonce_hashes, one was generated by a correct server, which implies that a client invoked a write with this timestamp (i.e., the write was not fabricated by faulty servers) and so it is eligible to be examined further. Otherwise, the client waits for more nonces before trying to decode a block.

If enough nonces are found, the client tries to reconstruct the block. A block can always be decoded given m fragments consistent with the fingerprinted cross-checksum (line 1817). If any fragments were provided with an additional cross-checksum value cc, the client can reconstruct a block and check if the erasure-coding of that block includes m fragments consistent with the fingerprinted cross-checksum. Since all correct servers will produce the same cross-checksum in **s_rpc_prepare_block**, it suffices to check each value of cc in turn (line 1818). If m fragments were returned with the same value cc or are consistent with the fingerprinted cross-checksum, the client decodes a block (line 1821). If at least m fragments in the erasure-coding of this block are consistent with the fingerprinted cross-checksum (lines 1822-1827), this block will be returned. Note that the check in **c_find_block** (lines 1822-1827) is identical to that in **s_rpc_prepare_block** (lines 1302–1307).

If a block is found, it is returned as the response of the read (line 1626). To ensure that this block is seen by subsequent reads, the client writes back fragments as needed (line 1625). In practice, the client can skip write back if any $2f+1$ of the first $3f+1$ servers claim to have committed this timestamp or a more recent one.

4.2.4 Correctness

This section provides arguments for the safety and liveness properties of the protocol.

Liveness

This section argues the liveness properties of write and read operations. Two notions of liveness are considered, namely *wait freedom* [49] and *obstruction freedom* [50]. Informally, an operation is wait-free if the invoking client can drive the operation to completion in a finite number of steps, irrespective of the behavior of other clients. An operation is obstruction-free if the invoking client can drive the operation to completion in a finite number of steps once all other clients are inactive for sufficiently long. That is, an obstruction-free operation may not complete, but only due to continual interference by other clients.

THEOREM 4.2.1. Write operations are wait-free.

Proof. **c_dowrite** invokes **s_rpc_prepare_frag** (line 1106) or **s_rpc_prepare_block** (line 1107) at each server. Each such call at a correct server returns successfully (line 1203 or 1309), implying that the client receives at least $n-f \geq 2f+1$ responses. Consequently, if ts as input to **c_dowrite** is NULL then the largest ts returned by servers is chosen (line 1111) and *found_largest_ts* is signaled (line 1112). An **s_rpc_prepare_frag** (line 1116) or **s_rpc_prepare_block** (line 1117) call is then placed at each server that did not return this ts. If ts as input to **c_dowrite** is not NULL, all correct servers will return this ts. By the time the last of the threads that will reach line 1120 does so (if not sooner), all correct servers have contributed a response for the same timestamp to *Prepare* from **s_rpc_prepare_frag** or **s_rpc_prepare_block**, causing *prepare_ready* to be signaled. The collected set of prepare responses in *Prepare* is then sent to all servers in an **s_rpc_commit** (line 1128). Because at least $m+f$ of the prepare responses in *Prepare* are from correct servers, at least $m+f$ of the prepare responses contain correct MAC values (line 1501) and so *Nonces* will include at least $m+f$ tuples (line 1503). Hence, these **s_rpc_commit** calls to correct servers return SUCCESS (line 1508), and so the write operation completes (line 1130). □

DEFINITION 4.2.2. If S_i.**s_rpc_commit**$(\mathsf{ts},\mathsf{fpcc},\mathit{Prepare})$ returns SUCCESS, then this commit at S_i is said to *rely on* S_j if $\mathit{Prepare}[j] = \langle \mathsf{ts}, \mathsf{nonce}, \langle \mathsf{tag}_k \rangle_{1 \leq k \leq n} \rangle$ and $\mathsf{tag}_i = \mathsf{MAC}_{j,i}(\langle \mathsf{ts}, \mathsf{fpcc}, \mathsf{nonce} \rangle)$.

LEMMA 4.2.3. In a correct client's **c_read**, suppose that for a fixed $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ the following occurs: From some correct S_i that previously returned SUCCESS to **s_rpc_commit**(ts, fpcc, $*$), and from each of m correct servers S_j on which the first such commit at S_i relies, the client receives $\langle \mathsf{data}, * \rangle$ in response to an **s_rpc_read**(ts, fpcc) call on that server (line 1620) where data $\neq$ ⟨NULL, NULL, NULL, NULL⟩. Then, the call to **c_find_block** (line 1621) including the last such response returns a block B $\neq$ NULL.

Proof. Consider such a $\mathsf{data} = \langle \mathsf{d}, \mathsf{cc}, \mathsf{nonce\_hash}, \mathit{Nonces} \rangle$ received from a correct server S_j. d either is consistent with fpcc as verified by S_j in **s_rpc_prepare_frag** (lines 1200–1202) and verified by the client in **c_find_block** (lines 1800–1803), or cc matches this fragment, as generated by S_j in **s_rpc_prepare_block** (lines 1304 and 1308) and verified by the client in **c_find_block** (line 1806). In the latter case, the fact that S_j reached line 1404 (where it saved $\langle \mathsf{d}, \mathsf{cc}, \mathsf{nonce\_hash}, * \rangle$) implies that previously $\mathsf{cnt} \geq m$ in line 1307, and so by the properties of fingerprinted cross-checksums (Theorem 2.2.4), all such servers received the same input block B in calls **s_rpc_prepare_block**(B, ts, fpcc) and so constructed the same cc in **s_rpc_prepare_block**.

Now consider the response $\mathsf{data} = \langle \mathsf{d}, \mathsf{cc}, \mathsf{nonce\_hash}, \mathit{Nonces} \rangle$ from the correct server S_i. Recall that S_i previously returned SUCCESS to **s_rpc_commit**(ts, fpcc, $*$), and that the first such commit relied on the m correct servers S_j. Since the focus is on the first such commit at S_i, and since data $\neq$ ⟨NULL, NULL, NULL, NULL⟩, the SUCCESS response was generated in line 1508, not 1500. In this case, *Nonces* $\neq$ NULL by lines 1503 and 1505, and in fact includes $\langle j, \mathsf{nonce} \rangle$ pairs for the m

correct servers S_j on which this commit relies. When the last data from S_i and these m servers S_j is passed to **c_find_block** (line 1621), each $\mathsf{nonce\_hash}$ present in each S_j's data will have a matching nonce in S_i's *Nonces*, i.e., such that $\mathsf{nonce\_hash} = \mathsf{hash}(\mathsf{nonce})$. Hence, $\mathsf{npreimages} \geq m \geq f+1$ (lines 1810–1813), and so the client will try to decode a block in lines 1815–1828.

If the responses from the m servers S_j on which the commit at S_i relies have $\mathsf{cc} = \text{NULL}$, a block is decoded and returned (line 1817). Otherwise, the client eventually tries to decode these fragments accompanied by $\mathsf{cc} = \text{NULL}$ (the set *Frags*) together with those accompanied by $\mathsf{cc} \neq \text{NULL}$ provided by these correct servers S_j (the set *Frags′*); see line 1821. Each S_j contributing a fragment of the latter type verified that fpcc was consistent with m fragments derived from the block B input to **s_rpc_prepare_block** (lines 1302–1307), and generated its fragment to be a valid fragment of B. Each fragment of the former type was verified by S_j to be consistent with fpcc (lines 1200–1202), and so is a valid fragment of B (with overwhelming probability by Corollary 2.1.12). Consequently, upon decoding any m of these fragments, the client obtains B, will find m fragments of the resulting block to be consistent with fpcc (lines 1822–1827), and so will return B (line 1828). □

THEOREM 4.2.4. The read protocol is obstruction-free.

Proof. In **c_read**, a call to **s_rpc_find_timestamp** is made to servers $1, \ldots, 3f+1$ (line 1605), to which each of at least $2f+1$ correct servers responds with its value of $\mathsf{latest\_commit}$ (line 1700). Consider the greatest timestamp $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ returned by a correct server, say S_i. This timestamp is greater than or equal to the timestamp from at least the $2f+1$ correct servers that responded (checked in lines 1612–1613), so the client tries to read fragments via **s_rpc_read** at this timestamp (line 1620). This timestamp was previously committed by S_i, as a correct server updates $\mathsf{latest\_commit}$ only in **s_rpc_commit** at line 1507. Moreover, this commit relies on at least m correct servers S_j; see line 1503. Now consider the following two possibilities for each of these correct servers S_j on which the commit relies:

- S_j assigned to $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in line 1404 because the condition in line 1402 evaluated to true, and has not subsequently deleted $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in line 1506. In this case, S_j returns the contents of $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in response to the **s_rpc_read** call (line 1902).

- S_j either did not assign to $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in line 1404 because the condition in line 1402 evaluated to false, or deleted $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in line 1506. In this case, $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle] =$ NULL and $\mathsf{latest\_commit} > \langle \mathsf{ts}, \mathsf{fpcc} \rangle$ in line 1900 (due to lines 1402 and 1507), and so S_j returns $\mathsf{latest\_commit}$ in response to the **s_rpc_read** call (line 1901).

If all m correct servers S_j fall into the first case above, then one of the client's calls to **c_find_block** (line 1621) returns a non-NULL block (Lemma 4.2.3). The client writes this with a **c_dowrite** call (line 1625), which is wait-free (Theorem 4.2.1), and then completes the **c_read**. If some S_j falls into the second case above, then the timestamp it returns (or a higher one returned by another correct server) satisfies the condition in lines 1612–1613 and so the client will subsequently read at this timestamp (line 1620) if a non-NULL block is not first returned from **c_find_block** (line 1621).

Consequently, for the client to never return a block in a **c_read**, correct servers must continuously return increasing timestamps in response to **s_rpc_read** calls. If there are no concurrent commits, then the client must reach a timestamp at which it returns a block. Hence, the read protocol is obstruction-free. □

Linearizability

Informally, linearizability [51] requires that the responses to read operations are consistent with an execution of all reads and writes in which each operation is performed at a distinct moment in real time between when it is invoked and when it completes. Only reads by correct clients need be considered, because no guarantees are provided to faulty clients. Since writes by faulty clients can be read by correct clients, however, such writes cannot be ignored. Consequently, the execution of **s_rpc_prepare_frag** or **s_rpc_prepare_block** by a faulty client at a correct server that returns $\langle \mathsf{ts}, \mathsf{nonce}, * \rangle$ (i.e., returns a value on line 1405 rather than returning FAILURE on line 1204 or 1310) is defined as to instantiate a write invocation at the beginning of time. The timestamp of the invocation is the pair $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ used to generate a nonce (line 1401). Each operation by a correct client also gets an associated timestamp $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$. For a write operation, the timestamp is that sent to **s_rpc_commit**. For a read operation, the timestamp is that of the write operation from which it read.

Proving that faulty write operations and correct write and read operations are linearizable shows that the protocol guarantees a natural extension to linearizability, limiting faulty clients to invoking writes that they could have invoked anyway at similar expense had they followed the protocol. The following five lemmas are used to prove that such a history is linearizable.

LEMMA 4.2.5. A read will share a timestamp with a write that has been invoked by some client.

Proof. Per line 1813, a call to **c_find_block**($*$, *State*, ts, fpcc, $*$) returns a non-NULL value only if *State*[$\langle$ts, fpcc$\rangle$], possibly modified per line 1809, includes a nonce_hash from some correct S_j such that some S_i.**s_rpc_read**(ts, fpcc) returned data $= \langle *, *, *, \mathit{Nonces} \rangle$, $\mathit{Nonces} \ni \langle j, \mathsf{nonce} \rangle$ and hash(nonce) = nonce_hash (data was passed to this or a previous **c_find_block**($*$, *State*, ts, fpcc, $*$) call, see lines 1620–1621, and then added to *State*[$\langle$ts, fpcc$\rangle$] in line 1809). This nonce was created by S_j on line 1401 with $\langle$ts, fpcc$\rangle$. Since a correct writer keeps each nonce secret unless it is returned from **s_rpc_prepare_frag** or **s_rpc_prepare_block** for the timestamp on which it settles for its write timestamp, this nonce shows that the writer, if correct, adopted $\langle$ts, fpcc$\rangle$ as its timestamp; consequently, the write with this timestamp was invoked, satisfying the lemma. If no correct writer performed a write with timestamp $\langle$ts, fpcc$\rangle$, then the creation of nonce by S_j in line 1401 with $\langle$ts, fpcc$\rangle$ implies that the write with timestamp $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ was invoked by a faulty client. □

LEMMA 4.2.6. Consider two invocations

$$\begin{aligned} \mathsf{B} &\leftarrow \textbf{c\_find\_block}(*, *, \mathsf{ts}, \mathsf{fpcc}, *) \\ \mathsf{B}' &\leftarrow \textbf{c\_find\_block}(*, *, \mathsf{ts}, \mathsf{fpcc}, *) \end{aligned}$$

at correct clients for the same timestamp $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$. If $\mathsf{B} \neq$ NULL and $\mathsf{B}' \neq$ NULL, then $\mathsf{B} = \mathsf{B}'$ with all but negligible probability.

Proof. A block $\mathsf{B} \neq$ NULL is returned by **c_find_block**$(*, *, \mathsf{ts}, \mathsf{fpcc}, *)$ at either line 1817 or line 1828. B is returned at line 1828 only if at least m erasure-coded fragments produced from B are consistent with fpcc, as checked in lines 1822–1827. Similarly, B is returned at line 1817 only after it is reconstructed from at least m fragments d' such that $\langle *, \mathsf{d}', \mathsf{cc}, *, * \rangle \in State[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ and $\mathsf{cc} =$ NULL; each such d' was confirmed to be consistent with fpcc in lines 1800–1803, in either this or an earlier invocation of the form **c_find_block**$(*, *, \mathsf{ts}, \mathsf{fpcc}, *)$. In either case, B has at least m erasure-coded fragments consistent with fpcc. If blocks B and B' each have at least m erasure-coded fragments that are consistent with the same fpcc, they are the same with all but negligible probability (Theorem 2.2.4). □

Lemma 4.2.6 states that two correct clients who read blocks at the same timestamp read the same block, since the block returned from **c_read** is that produced by **c_find_block** (lines 1621–1626).

Lemmas 4.2.7–4.2.9 show that timestamp order for operations is consistent with real-time precedence.

LEMMA 4.2.7. Consider two write operations performed by correct clients. If the response to one precedes the invocation of the other, then the timestamp of the former is less than the timestamp of the latter.

Proof. Before the earlier write returns a response in line 1130, at least $n - f$ servers returned SUCCESS from **s_rpc_commit**$(\mathsf{ts}, \mathsf{fpcc}, Prepare)$, where $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ is the timestamp of this write. In doing so, at least $m = n - 2f$ correct servers record $\mathsf{latest\_commit} \leftarrow \langle \mathsf{ts}, \mathsf{fpcc} \rangle$ (line 1507) if not greater (line 1500). Consequently, the ts value returned in the prepare phase for the later write by these servers will be greater than the ts value for the earlier write (line 1400). Because there are $n = m + 2f$ total servers, any $2f + 1$ servers will include one of these m correct servers, so the ts chosen (line 1111) will be greater than the ts value in the timestamp for the earlier write. □

LEMMA 4.2.8. Consider a write operation and a read operation, both performed by correct clients. If the response to the write precedes the invocation of the read, then the timestamp of the write is at most the timestamp of the read.

Proof. Before the write returns a response in line 1130, at least $n - 2f$ correct servers returned SUCCESS from **s_rpc_commit**$(\mathsf{ts}, \mathsf{fpcc}, Prepare)$, where $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ is the timestamp of this write. In doing so, at least $n - 2f$ correct servers record $\mathsf{latest\_commit} \leftarrow \langle \mathsf{ts}, \mathsf{fpcc} \rangle$ (line 1507) if not greater (line 1500). These $n - 2f = m$ correct servers include at least $f + 1$ of the servers $1, \ldots, 3f + 1$, and so in the read operation at most $2f$ of servers $1, \ldots, 3f + 1$ respond to **s_rpc_find_timestamp** at line 1700 with a lower timestamp than $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$. Because a read considers only timestamps that are at least as large as those returned by $2f + 1$ of the first $3f + 1$ servers (line 1613), the read will be assigned a timestamp at least as large as $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$. □

LEMMA 4.2.9. If the response of a read operation by a correct client precedes the invocation of another (write or read) operation by a correct client, then the timestamp of the former operation is at most the timestamp of the latter.

Proof. A read calls **c_dowrite** at its timestamp before returning a response (line 1625). This will have the same affect as completing a write at that timestamp. Consequently, the later operation will have a higher timestamp if it is a write (Lemma 4.2.7) and a timestamp at least as high if it is a read (Lemma 4.2.8). □

THEOREM 4.2.10. Write and read operations are linearizable.

Proof. To show linearizability, construct a linearization (total order) of all read operations by correct clients and all write operations that is consistent with the real-time precedence between operations such that each read operation returns the block written by the preceding write operation. First, order all writes in increasing order of their timestamps. By Lemma 4.2.7, this ordering does not violate real-time precedence. Next, place all read operations with the same timestamp immediately following a write operation with that timestamp, ordered consistently with real-time precedence (i.e., each read is placed somewhere after all other operations with the same timestamp that completed before it was invoked). By Lemma 4.2.5, each read is placed after some write operation. This placement does not violate real-time precedence with the next (or any) write operation in the linearization, since if the next write had completed before this read began, then by Lemma 4.2.8 this read operation could not have the timestamp it does. Real-time precedence between reads with different timestamps cannot be violated by this placement, by Lemma 4.2.9. Since all reads with the same timestamp read the same block (Lemma 4.2.6)—which is the block written in the write with that timestamp if the writer was correct—write and read operations are linearizable. □

4.3 Implementation

The prototype is evaluated and compared to competing approaches using a distributed storage prototype. The low-overhead fault-tolerant prototype consists of a client library, linked to directly by client applications, and a storage server application. The prototype supports the protocol described in Section 4.2 as well as several competing protocols, as described in Section 4.4.1.

The client library interface consists of two functions, "read block" and "write block." In addition to the parameters described in Section 4.2, read and write accept a block number as an additional argument, which can be thought of as running an instance of the protocol in parallel for every block in the system. Each server in a pool of storage servers runs the storage server application, which accepts incoming RPC requests and executes as described in Section 4.2. Clients and servers communicate with remote procedure calls over TCP sockets. Each server has a large NVRAM cache, where non-volatility is provided by battery backup. This allows most writes and many reads to return without disk I/O.

The prototype uses 16 byte fingerprints generated with the evaluation homomorphic fingerprinting function (Section 2.4). Fingerprinting is fast, and only the first m fragments are fingerprinted (the other fingerprints can be computed from these fingerprints). Homomorphic fingerprinting requires a small random value for each distinct block that is fingerprinted. This random value is provided by a hash of the cross-checksum in the protocol. After computing this random value, the implementation precomputes a 64 kB table, which takes about 20 microseconds. After computing this table, fingerprinting each byte requires one table lookup and a 128-bit XOR. This implementation can fingerprint about 410 megabytes per second on a 3 GHz Pentium D processor.

The client library uses Rabin's Information Dispersal Algorithm [84] for erasure coding. The first m fragments consist of the block divided into m equal fragments (that is, a systematic encoding is used). Since $m > f$ and only $m + f$ fragments must be encoded, this cuts the amount of encoding by more than half.

The SHA-1 and HMAC implementations from the Nettle toolkit [76] is used, which can hash about 280 megabytes per second on a 3 GHz Pentium D processor. Each hash value is 20 bytes, and each MAC is 8 bytes. Due to ongoing advances in the cryptanalysis of SHA-1 [33], a storage system with a long expected lifetime may benefit from a stronger hash function. The performance of SHA-512 on modern 64-bit processors has been reported as comparable to that of SHA-1 [42]. Hence, though not measured, the prototype should achieve similar performance if the hash were upgraded on such systems.

The prototype implements a few simple optimizations for the protocol. For example, during commit, only the appropriate pairwise MAC is sent to each server. The client tries writing at the first $m + f$ servers, considering other servers only if these servers are faulty or unresponsive. Similarly, the client requests timestamps from the first $2f + 1$ servers and reads fragments from the first m servers, which allows most reads to return in a single round of communication without requiring any decoding beyond concatenating fragments. Furthermore, if $f + 1$ or more servers return matching timestamps, the client does not request nonces because it can conclude that a correct server committed this write and hence some client invoked a write at this timestamp, satisfying Lemma 4.2.5.

Also, the client limits the amount of state required for a read operation by considering only the most recent timestamp proposed by each server. It does not consider more timestamps until all but f servers have returned a response for all timestamps currently under consideration. Servers delay garbage collection by a few seconds, thus obviating the need for fast clients to ever follow garbage collection redirection.

4.4 Evaluation

This section evaluates the protocol proposed in this chapter (the FP protocol) on a distributed storage prototype. Three competing protocols that are described in Section 4.4.1 were also implemented and evaluated. The experimental setup is described in Section 4.4.2. Single client write throughput, read throughput, and response time are evaluated in Sections 4.4.3, 4.4.4, and 4.4.5, respectively. An analysis of the protocol presented in Section 4.2 suggests that the protocol should perform similarly to a benign erasure-coded protocol with the additional computational expense of hashing and the extra bandwidth required for the fpcc, MACs, and nonces. The experimental results confirm that the FP protocol is competitive with the benign erasure-coded protocol and that it significantly outperforms competing approaches.

4.4.1 Competing Protocols

To enable fair comparison, the distributed storage prototype supports multiple protocols. Protocols are compared within the same framework to ensure that measurements reflect protocol variations rather than implementation artifacts. The following three protocols are evaluated in addition to the FP protocol.

Benign erasure-coded protocol: The prototype implements an erasure-coded storage protocol that tolerates crashes but not Byzantine faulty clients or servers. This protocol uses the same erasure coding implementation used for the FP protocol. A write operation encodes a block into $m+f$ fragments and sends these fragments to servers. A read operation reads from the first m servers, avoiding the need to decode. Both complete in a single round of communication. This protocol assumes concurrency is handled by some external locking protocol; a real implementation would require more rounds of communication for some writes, but this overhead is ignored. Hence, this protocol provides an upper bound for the performance of any erasure-coded fault-tolerant storage protocol (Byzantine or not).

Benign replication-based protocol: The prototype implements a replication-based storage protocol that tolerates crashes but not Byzantine faulty clients or servers. A write operation sends the block to $f+1$ servers, and a read operation reads from a single server. As in the benign erasure-coded protocol, this protocol assumes concurrency is handled by an external locking protocol. Hence, though this protocol does not tolerate Byzantine faults, it represents an upper bound for the performance of any replication-based Byzantine (or not) fault-tolerant storage protocols. It is worth noting, however, that this implementation is substantially faster than most replication-based Byzantine fault-tolerant storage protocols found in the literature, which often require all-to-all broadcasts [18, 23], public-key signatures [19, 66], or writing to $2f+1$ or more replicas [18, 23].

Replication-based protocols are excellent for reads, but write performance is reduced due to bandwidth limitations. One method to overcome the network bandwidth limitation between the client and the switch for a single client is to use network-level multicast. The replication-based protocol does not use multicast for several reasons. Multicast is unavailable, unsuitable, or unstable in many network environments [44], and retransmissions due to congestion cannot take advantage of multicast. Also, multicast does nothing to reduce disk bandwidth and network bandwidth between the switch and the servers. Though each server could encode its own block to reduce disk bandwidth [18], a multicast-based protocol would not scale when multiple clients are writing to the same storage servers.

m+3f Byzantine fault-tolerant erasure-coded protocol: An alternative to Byzantine fault-tolerant replication-based storage is Byzantine fault-tolerant erasure-coded storage. The prototype implements a protocol similar to PASIS [44]. PASIS was engineered to improve server throughput by offloading work to clients. This protocol uses the same erasure coding implementation and SHA-1 library used for the FP protocol. The protocol implemented by the prototype, however, only emulates a PASIS-like protocol. It does not implement the versioning storage required by PASIS, nor does it run a garbage collection protocol, and, hence, it would not be suitable for storing data in a Byzantine environment. This implementation does, however, provide a comparison point against the approach most similar to the FP protocol.

To write a block, the client requests the most recent timestamp from each server. It then encodes the block into $m+3f$ fragments and hashes each fragment to create a cross-checksum. (Because a systematic encoding is used, "encoding" the first f fragments does not require computation.) By comparison, the FP protocol encodes and hashes $m+f$ fragments; f fewer because there are f fewer servers and another f fewer due to the partial encoding optimization, described in Section 4.2.1. To complete the write, the $m+3f$ protocol sends fragments to the first $m+2f$ servers, considering other servers only if some servers are unresponsive. By comparison, the FP protocol and the benign erasure-coded protocol send f fewer fragments because they need f fewer servers. To read a block, the client requests fragments from the first m servers along with timestamps from the first $3f+1$ servers. Assuming all timestamps match, the client must then verify the cross-checksum, which is embedded in the timestamp. This requires repeating the write computation: the client must encode and hash $m+3f$ fragments to recompute the cross-checksum.

4.4.2 Experimental Setup

All experiments are measured using a single client and a collection of servers. Each machine has a dual-core 3 GHz Pentium D processor, 2 GB of RAM, and an Intel PRO/1000 Gigabit Ethernet controller, and machines run Linux kernel version 2.6.18. Measurements are taken in the absence of concurrency and faults, which is expected to be the normal mode of operation in such a storage system. The client and the servers are connected to the same HP ProCurve Switch 2848 with QoS passthrough mode set to one-queue and flow control enabled for each port. Each experiment was run 10 times for 60 seconds, with the average performance reported in the figures. Standard deviations are all within 2% of the average, and performance matches analytical expectations.

The working set of data for each experiment is chosen to fit within the server caches, and the client does not cache data. The data gets loaded before measurements, ensuring 100% read hits, and the servers use write-back with synchronizing to disk disabled. The systems are battery-backed, but the experimental reason for this setup is to allow the measurement of protocol overhead rather than disk latency. Avoiding disk accesses makes performance dependent on the network and computational behavior of the protocols. If the working set does not fit in server caches, or if durability requirements prevent using NVRAM for write-back, the choice of protocol matters less because system performance will be limited by disk performance. In such a scenario, there is an even stronger argument for using a Byzantine fault-tolerant protocol rather than a protocol that tolerates only crash faults.

The client benchmark program is run on a single machine. It generates a synthetic workload. For throughput measurements, the client spawns several parallel threads, each of which issues a read or write request for a randomly selected 64 kB block, waits for the response, and then issues another request. For response time measurements, a single thread issues a single write request, waits for a response, and then repeats. For erasure-coded protocols (all but replication), $m = f+1$. Because the block size is fixed, the fragment size for erasure-coded protocols decreases as m increases (i.e., fragment size is $64/m$ kB).

4.4.3 Write Throughput

Figure 4.4.4 shows the write throughput achieved by a single client executing each of the four protocols as a function of the number of faults tolerated. The FP protocol significantly outperforms the Byzantine fault-tolerant $m+3f$ erasure-coded protocol as well as the crash fault-tolerant replication-based protocol, and it nearly matches the performance of the benign erasure-coded protocol. For example, at $f=4$, the FP protocol achieves a factor of 2.6 higher throughput than replication, a

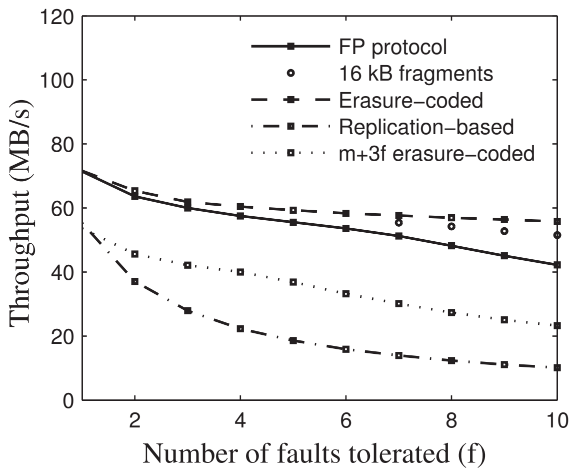


Figure 4.4.4: Write throughput for each protocol as a function of faults tolerated. The lines report the performance of each protocol when writing 64 kB blocks. The four circles report the performance of the FP protocol when writing 16 kB fragments (rather than $64/m$ kB).

factor of 1.4 higher throughput than the $m+3f$ protocol, and is within 5% of the performance of the erasure-coded protocol that does not tolerate Byzantine faulty servers or clients.

Each protocol requires a different number of servers to tolerate the same number of faults. The benign erasure-coded protocol and the FP protocol require $m+f$ responsive servers, the replication-based protocol requires $f+1$ servers, and the $m+3f$ protocol requires $m+2f$ responsive servers. (Both Byzantine fault-tolerant protocols must be able to reach an additional f servers if some of these servers are not responsive.) For example, for $f=4$ and $m=5$, the benign erasure-coded protocol and the FP protocol write data to 9 servers, the replication-based protocol writes to 5 servers, and the $m+3f$ protocol writes to 13 servers.

Throughput is the amount of useful data written, which is less than the amount of data sent over the network. The FP protocol and the benign erasure-coded protocol both send $\frac{|\mathsf{B}|}{m}(m+f)$ bytes when writing a block of $|\mathsf{B}|$ bytes. Replication must send $|\mathsf{B}|(f+1)$ bytes, and the $m+3f$ protocol must send $\frac{|\mathsf{B}|}{m}(m+2f)$ bytes. The erasure-coded protocols could increase throughput for constant f by increasing m beyond $f+1$.

The benign erasure-coded protocol performs well, as expected, achieving a write throughput close to $\frac{m}{m+f}$ of the total network bandwidth available. The FP protocol performs almost as well. When tolerating up to 6 Byzantine faulty servers, it performs within 10% of the benign protocol that only tolerates server crashes. As the number of servers in the system grows, however, the additional network overhead in the FP protocol becomes noticeable for two reasons. First, because block size is constant, the size of the fragment written at each server decreases as the number of servers increases. Second, the sizes of the fpcc, MACs, and nonces increase as the number of servers increases. For $f=6$, fragment size is over 9 kB and fpcc, MAC, and nonce overhead is under 700 bytes (overhead is under 7% of data sent). For $f=10$, fragment size is under 6 kB and fpcc, MAC, and nonce overhead is over 1100 bytes (overhead is over 15% of data sent).

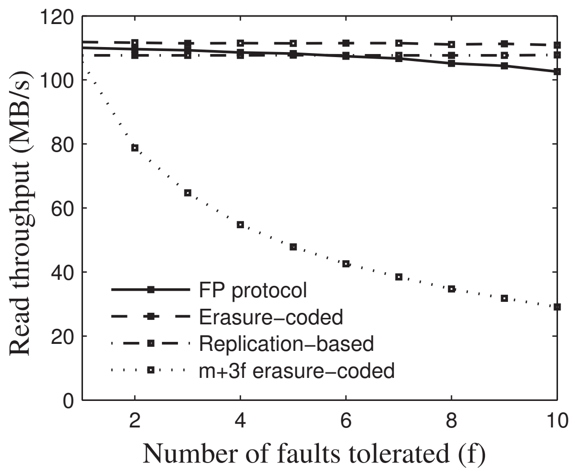


Figure 4.4.5: Read throughput as a function of faults tolerated.

One solution to this problem is to increase the block size. For example, the four circles in Figure 4.4.4 show the throughput of the FP protocol when fragment size is 16 kB. The FP protocol performs within 10% of the benign protocol when the fragment size is increased to 16 kB for both protocols, even when tolerating 10 faults. (The benign protocol performs less than 3% better when the fragment size is increased to 16 kB.)

The replication-based protocol performs poorly for all but the smallest number of faults tolerated, as expected, because it writes $(f+1)/(\frac{m+f}{m}) > (f+1)/2$ times as much data as the benign erasure-coded protocol. The $m+3f$ Byzantine fault-tolerant erasure-coded protocol writes $\frac{m+2f}{m+f} \approx 1.5$ times as much data as the benign erasure-coded protocol, and up until about $f=4$ it is only a factor of 1.5 times worse. The $m+3f$ protocol, however, must encode and hash $m+3f$ fragments to generate the cross-checksum even though it only writes to $m+2f$ servers because it does not include the partial encoding optimization. Hence, for $f>4$, the $m+3f$ protocol is computationally bound by the client.

4.4.4 Read Throughput

Figure 4.4.5 shows the read throughput achieved by a single client executing each of the four protocols as a function of the number of faults tolerated. The FP protocol achieves read throughput within 10% of the two benign protocols and significantly outperforms the $m+3f$ protocol. The slight drop for the FP protocol and the benign erasure-coded protocol as the number of faults tolerated increases is due to network congestion caused by the increasing number of servers providing responses.

All four protocols read the same amount of data. The replication-based protocol reads an entire 64 kB block from a single server, and the other protocols read fragments from m servers. The erasure-coded protocols read from the first m servers to avoid the need to decode. In addition to fragments, the Byzantine fault-tolerant protocols must read timestamps from more servers to check for concurrency. The FP protocol reads timestamps from $2f+1-m=f$ more servers, while the

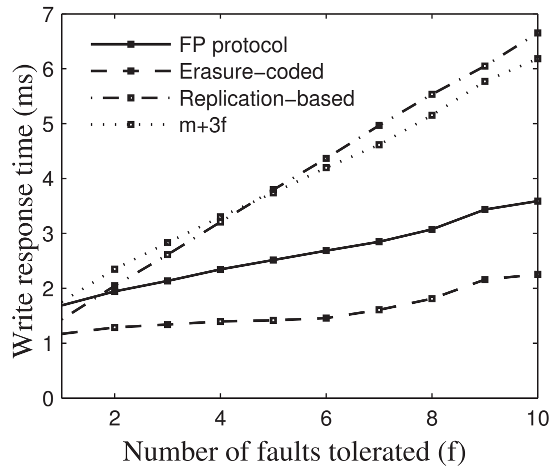


Figure 4.4.6: Write response time as a function of faults tolerated.

	RPC	Encode	Hashing	Fingerprinting
Erasure coded	1.46 ms	0.79 ms	–	–
Replication	6.65 ms	–	–	–
FP protocol	2.17 ms	0.79 ms	0.45 ms	0.18 ms
$m+3f$	2.80 ms	2.54 ms	0.88 ms	–

Figure 4.4.7: Write response time breakdown for $f = 10$.

$m+3f$ protocol reads timestamps from $3f+1-m = 2f$ more servers. Assuming all timestamps match, read completes in a single round of communication.

Once fragments are read, the Byzantine fault-tolerant protocols must verify data. The FP protocol requires just a hash and a fingerprint of the fragments. The $m+3f$ protocol, however, must recompute the cross-checksum, which requires encoding and hashing $m+3f$ fragments and is quite expensive for large values of f.

4.4.5 Response Time

Figure 4.4.6 shows the response time of a single write for each of the four protocols as a function of the number of faults tolerated. The FP protocol requires on average 1.04 ms more to complete a write operation than the benign erasure-coded protocol, which is on average 1.65 times worse. This is, however, a substantial improvement over the $m+3f$ protocol and the replication-based protocol, which both scale worse than the FP protocol.

Figure 4.4.7 provides a breakdown of the average latency of each operational component of a write for $f = 10$ as seen by the client. The table lists the time each protocol spent encoding, hashing, and fingerprinting fragments (other computational contributions were negligible); it also lists the time spent waiting for the network, which includes time spent in the kernel. As seen in the table, about half of the additional latency for a write by the FP protocol as compared to the benign erasure-coded protocol is due to hashing and fingerprinting, and the other half is due to the extra

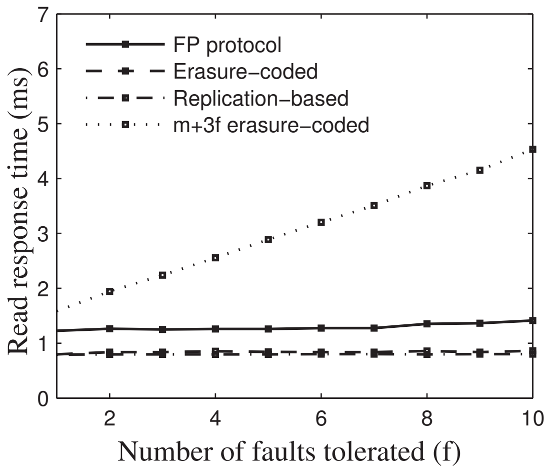


Figure 4.4.8: Read response time as a function of faults tolerated.

	RPC	Encode	Hashing	Fingerprinting
Erasure coded	0.86 ms	–	–	–
Replication	0.80 ms	–	–	–
FP protocol	0.97 ms	–	0.24 ms	0.20 ms
$m+3f$	1.10 ms	2.54 ms	0.90 ms	–

Figure 4.4.9: Read response time breakdown for $f = 10$.

round of communication. The additional latency for the replication-based protocol is, of course, due to the extra bandwidth required to write f replicas. The additional latency for the $m+3f$ protocol is due to the encoding of $2f$ more fragments, the extra round of communication, the sending of 1.5 times as many fragments, and the hashing of $m+3f$ fragments.

Read response time (Figure 4.4.8) is as expected. Table 4.4.9 provides a breakdown of the average latency of each operational component of a read for $f = 10$ as seen by the client. The benign erasure-coded protocol and the replication protocol require on average 0.84 ms and 0.80 ms respectively to read a single block when tolerating between one and ten faults. The FP protocol requires on average 1.29 ms, the difference being the time needed to hash and fingerprint fragments. Each of these protocols requires about the same amount of time to read a block when tolerating one fault as when tolerating ten faults. The $m+3f$ protocol requires 3.05 ms on average, and it requires 1.58 ms to read a single block when tolerating one fault but 4.54 ms when tolerating ten faults. It scales worse than the other protocols because it must encode and hash $m+3f$ fragments to recompute the cross-checksum.

4.5 Conclusion

Distributed block storage systems can tolerate Byzantine faults in asynchronous environments with little overhead over systems that tolerate only crashes. Replication-based block storage protocols are effective for workloads that are mostly reads or when tolerating a single fault, but exhibit low throughput and high latency for large writes. Erasure-coded protocols provide higher throughput writes and can increase m for fixed f to realize even higher throughput. Previous Byzantine fault-tolerant erasure-coded protocols, however, exhibit low client throughput for reads and high computational overheads for both reads and writes. This chapter presents the FP protocol, a Byzantine fault-tolerant erasure-coded protocol that performs well for both reads and large writes. Measurements of a prototype implementation demonstrate that this protocol exhibits throughput within 10% of the ideal crash fault-tolerant erasure-coded protocol for reads and sufficiently large writes. Furthermore, the FP protocol has little computational overhead other than a cryptographic hash and a homomorphic fingerprint of the data.

Chapter 5

Scalable Fault Tolerance through Byzantine Locking

As distributed systems grow in size and importance, they must tolerate complex software bugs and hardware misbehaviors in addition to simple crashes and lost messages. Byzantine fault-tolerant protocols can tolerate arbitrary problems, making them an attractive building block—in theory. But, in practice, system designers continue to worry that their performance overheads and scalability limitations are too great. Recent research has improved performance by exploiting optimism to improve common cases, but a significant gap still exists.

The Zzyzx replicated state machine protocol bridges that gap with a new technique called *Byzantine Locking*. Layered atop a Byzantine fault-tolerant replicated state machine protocol (e.g., PBFT [24] or Zyzzyva [55]), Byzantine Locking can be used to temporarily give a client exclusive access to state in the replicated state machine. It uses the underlying Byzantine fault-tolerant replicated state machine to extract the relevant state and, later, to re-integrate it. Unlike locking in non-Byzantine fault-tolerant systems, Byzantine Locking is only a performance tool. To ensure liveness, locked state is kept on servers, and a client that tries to access objects locked by another client can request that the locks be revoked, forcing both clients back to the underlying replicated state machine to ensure consistency.

Byzantine Locking provides unprecedented scalability and efficiency for the common case of infrequent concurrent data sharing. Most notably, the server processes to which locked state is extracted for servicing operations by the locking client—in the parlance of Byzantine Locking, *log servers*—can execute on distinct physical computers from the replicas for the underlying replicated state machine. Thus, multiple log server groups, each running on distinct physical computers, can be used for independently locked state, allowing throughput to be scaled by adding computers, as shown in Figure 5.0.1. Even when running the log servers on the same computers as the underlying replicated state machine, exclusive access allows clients to execute a sequence of operations much more efficiently (just one round-trip with only $2f+1$ responses, where f if the number of faulty servers tolerated), because concurrency is explicitly precluded. This performance benefit can be seen in Figure 5.0.1 by comparing the 4-server throughputs of Zyzzyva and Zzyzx—Byzantine Locking enables the factor of $2.9\times$ higher throughput shown.

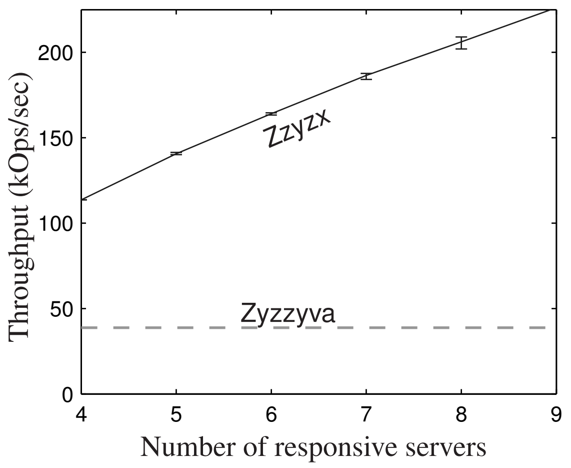


Figure 5.0.1: **Throughput vs. servers.** *Zzyzx's throughput scales nearly linearly as servers are added. Zyzzyva does not use additional servers to improve throughput, so the dashed line repeats its 4-server throughput for reference. Even with the minimum number of servers (4), Zzyzx significantly outperforms Zyzzyva. The measured workload includes no faults or data sharing among clients. All configurations measured tolerate one Byzantine fault (f=1). Section 5.5 describes the experimental setup and results in detail.*

Zzyzx implements Byzantine Locking on top of Zyzzyva [55], a state-of-the-art Byzantine fault-tolerant replicated state machine, to improve performance while providing the same correctness and liveness guarantees as Zyzzyva. Experiments, described in Section 5.5, show that Zzyzx can provide 39–43% lower latency and a factor of 2.2–2.9× higher throughput when using the same servers, compared to Zyzzyva, for operations on locked objects. Postmark [52] completed 60% more transactions on a Zzyzx-based file system than one based on Zyzzyva, and Zzyzx provided a factor of 1.6× higher throughput for a trace-based metadata workload. The benefits of locking outweigh the cost of unlocking after as few as ten operations. Operations on concurrently shared data objects do not use the Byzantine Locking layer—clients just execute the underlying Zyzzyva protocol directly. Thus, except when transitioning objects from unshared to shared, the common case (unshared) proceeds with maximal efficiency and the uncommon case is no worse off than the underlying Byzantine fault-tolerant replicated state machine.

Although it will provide correct behavior under any workload, the benefits of Byzantine Locking will be realized most in services whose state consist of many objects that are rarely shared. This characterization fits many critical services for which both scalability and Byzantine fault tolerance is desirable. For example, the metadata service of most distributed file systems contains a distinct object for each file or directory, and concurrent sharing is rare [10]. Similarly, a distributed key-value store for website personalization [26] or e-commerce shopping carts [34] may have many concurrent writers that access distinct keys.

This chapter makes three primary contributions. First, it introduces Byzantine Locking as a means of realizing unprecedented scalability and efficiency for Byzantine fault-tolerant replicated state machines when concurrent data sharing is uncommon. Second, it describes Zzyzx, a Byzantine fault-tolerant replicated state machine protocol that layers Byzantine Locking atop Zyzzyva to demonstrate the performance and scalability features of Byzantine Locking. Third, it uses the Zzyzx prototype to evaluate the performance characteristics of Byzantine Locking and shows that Byzantine Locking provides the expected scaling and significant performance improvements for collections of usually-independent objects.

5.1 Context and Related Work

Large distributed systems exhibit more faults and more types of faults than traditional fault-tolerance techniques can manage. For example, a single misdirected write can corrupt data in protocols such as Paxos. In response, many practitioners use more robust protocols. Modern systems include a variety of checksums and consistency checks, but ad-hoc robustness mechanisms do not capture some real failure modes [9, 25, 60]. Efficient protocols that survive the fuller range of failure modes are important for the critical services needed by cloud computing, data centers, and clustered storage systems, where the higher frequency of faults, wider diversity of corruptions and faults seen in practice, and asynchronous network behavior break traditional techniques.

As fault tolerance has grown more important and techniques to achieve fault tolerance less expensive, there has been a natural progression of distributed protocols used by practitioners, from unreplicated to replicated, synchronous to asynchronous, and crash-tolerant to ad-hoc consistency checks. The next step is from ad-hoc consistency to full Byzantine fault tolerance.

Byzantine fault-tolerant protocols tolerate any number of faulty or malicious clients and a fraction of faulty or malicious servers. Byzantine fault tolerance ensures that all bases are covered, protecting against misdirected writes, soft errors, and other faults and corruptions found in modern hardware and software. A Byzantine fault-tolerant replicated state machine protocol can be used to implement any deterministic service. Given the growth in size and importance of many distributed services, one would like to use Byzantine fault-tolerant replicated state machines to make such services more robust. Toward that end, much recent research has focused on designing Byzantine fault-tolerant replicated state machine protocols with improved performance and scalability, especially during fault-free periods of operation.

5.1.1 The Byzantine Efficiency Race

Recent years have seen something of an arms race among researchers seeking to provide application writers with efficient Byzantine fault-tolerant substrates. Perhaps unintentionally, Castro and Liskov [24] initiated this race in proposing a new protocol and labeling it "practical," because it demonstrated performance considerably better than most expected could be achieved with Byzantine fault-tolerant systems. Their protocol replaces the digital signatures common in previous protocols with message authentication codes (MACs) and also increases efficiency with request batching, link-level broadcast, and agreement-free optimistic reads [21]. Still, the protocol requires four message delays and all-to-all communication, for all mutating operations, leaving room for improvement. Castro and Liskov called their protocol "BFT". To avoid confusion with the acronym for Byzantine fault tolerance, this paper follows the convention of calling their protocol PBFT.

	PBFT	Q/U	HQ	Zyzzyva	**Zzyzx**	RSM Lower Bound
Total servers required	**3f+1**	5f+1	**3f+1**	**3f+1**	**3f+1**	3f+1 [101]
Responsive servers required	**2f+1**	4f+1	**2f+1**	3f+1	**2f+1**	2f+1
Bottleneck MAC ops per req.	2+(8f+1)/B	2+8f	4+4f	2+3f/B	**2**	1
Message delays per req.	4	**2**	4	3	**2**	2
Throughput scales with # servers	No	Some*	Some*	No	**Yes**	–

Figure 5.1.2: **Comparison of Byzantine fault-tolerant replicated state machine protocols in the absence of faults and contention, along with commonly accepted lower bounds.** Data for PBFT, Q/U, HQ, and Zyzzyva are taken from [55]. Bold entries identify best-known values. f denotes the number of server faults tolerated, and B denotes the request batch size (see Section 5.5). "Responsive servers needed" refers to the number of servers that must respond in order to achieve good performance. *The throughput scalability provided by quorum protocols is limited by the requirement for overlap between valid quorums [69, 71].

Abd-el-Malek et al. [2] proposed Q/U, a quorum-based Byzantine fault-tolerant protocol that exploits speculation and quorum constructions to provide throughput that can increase somewhat with addition of servers. Q/U provides Byzantine fault-tolerant operations (including multi-object operations) on a collection of objects comprised of state and associated operations. Operations are optimistically executed in just one round-trip, and object histories are used to resolve issues created by concurrency or failures. Fortunately, both are expected to be rare in many important usages. One example of this is the file servers that have been used as concrete examples in papers on this topic (e.g., [24]). Matching conventional wisdom, analysis of NFS traces from a departmental server [38] confirms that most files are used by a single client and that, when a file is shared, there is almost always only one client using it at a time. For example, in the traces examined, fewer than one in one thousand operations would experience any contention in a replicated state machine protocol. If one discounts the two most contentious file handles, less than one in ten thousand operations would experience contention on average.

Cowling et al. [32] proposed HQ, which uses a hybrid approach to achieve the benefits of Q/U without the increased minimum number of servers ($3f$+1 for HQ vs. $5f$+1 for Q/U). Optimistically, an efficient quorum protocol executes operations unless concurrency or failures are detected. Each operation that encounters such issues then executes a second protocol to achieve correctness. In reducing the number of servers, HQ increases the common case number of message delays for mutating operations to four (two roundtrips).

Most recently, Kotla et al. [55] proposed Zyzzyva, which avoids all-to-all communication without additional servers, performs better than HQ under contention, and requires only three message delays. Unlike other protocols, however, Zyzzyva requires that all $3f+1$ nodes are responsive in order to achieve good performance; timeouts trigger a second protocol phase. Unfortunately, whether by misfortune or by design, some servers in many distributed systems will sometimes respond more slowly than others. Furthermore, requiring that all $3f+1$ servers respond to avoid extra work precludes techniques that reduce the number of servers needed in practice. For example, if only $2f+1$ servers need be responsive, the f "non-responsive" servers can be shared by neighboring Byzantine fault-tolerant clusters or used to fill other needs without being burdened under normal operation. Non-responsive servers could even be turned off to save power, at the cost of higher latency when faults occur.

In a recent study of several Byzantine fault-tolerant replicated state machine protocols, Singh et al. concluded that "one-size-fits-all protocols may be hard if not impossible to design in practice" [96]. They note that "different performance trade-offs lead to different design choices within given network conditions." Indeed, there are several parameters to consider, including the total number of replicas, the number of replicas that must be responsive for good performance, the number of message delays in the common case, the performance under contention, and the throughput, which is roughly a function of the numbers of cryptographic operations and messages per request.

Unfortunately, none of the above protocols score well on all of these metrics, as shown in Figure 5.1.2. PBFT requires four message delays and all-to-all communication, Q/U requires additional replicas, HQ requires four message delays and performs poorly under contention, and Zyzzyva performs poorly unless all nodes are responsive. (Zyzzyva5, a variant of Zyzzyva, performs well even when some nodes are not responsive, but requires additional replicas [55].)

5.1.2 How Zzyzx Fits In

Like the systems above, Zzyzx is optimized to perform well in environments where faults are rare and concurrency is uncommon, while providing correct operation under harsher conditions. During benign periods, Zzyzx outperforms and scales better than all of the prior approaches, requiring the minimum possible numbers of message delays (two, which equals one round-trip), responsive servers ($2f$+1), and total servers ($3f$+1). Zzyzx provides unprecedented scalability, because it does not require overlapping quorums as in prior protocols (HQ and Q/U) that provide any scaling; non-overlapping server sets can be used for frequently unshared state. When concurrency is common, Zzyzx performs similarly to its underlying protocol (e.g., Zyzzyva).

Zzyzx takes inspiration from the locking mechanisms used by many distributed systems to achieve high performance in benign environments. For example, GPFS uses distributed locking to provide clients byte-range locks that enable its massive parallelism [92]. In benign fault-tolerant environments, where lockholders may crash or be unresponsive, other clients or servers must be able to break the lock. To tolerate Byzantine faults, the protocol must additionally ensure that lock semantics are not violated by faulty servers or clients and that a broken lock is always detected by correct clients. Section 5.3 details how this is accomplished for Byzantine Locking.

By allowing clients to acquire locks, and then only allowing clients that have the lock on given state to execute operations on it, Zzyzx achieves much higher efficiency for sequences of operations from that client. Each replica can proceed on strictly local state, given evidence of lock ownership, thus avoiding all inter-replica communication. Also, locked state can be can be transferred to other servers, allowing non-overlapping sets of servers to handle independently locked state.

Zzyzx handles concurrency in a similar, though reverse, manner to HQ. In particular, clients are not required to use the Byzantine Locking, but they can do so when concurrency is not expected. So, whereas HQ falls back on a protocol such as PBFT or Zyzzyva that handles concurrency well each time concurrency is discovered, Zzyzx can use PBFT or Zyzzyva natively until concurrency is determined to be uncommon, at which point clients begin using Byzantine Locking. Thus, Zzyzx can avoid performance losses under concurrency, if good policy decisions are made, while gaining performance and scalability for rarely-shared state.

5.1.3 Prior Byzantine Fault-Tolerant Replicated State Machine Protocols

Recent Byzantine fault-tolerant replicated state machine protocols, such as PBFT, Q/U, HQ, Zyzzyva, and Zzyzx built upon several years of prior distributed systems research [17, 36, 54, 63, 69, 86, 87]. For example, Reiter [86] proposed the Rampart toolkit, which implements an asynchronous Byzantine fault-tolerant replicated state machine. Rampart uses an atomic multicast protocol and a secure membership protocol. The atomic multicast protocol is similar to PBFT, except the primary in PBFT is called the *sequencer* in Rampart. Public-key cryptographic overhead limits the performance of Rampart.

Rampart's membership protocol highlights the importance of the system model. In Reiter's model, there are an unbounded number of servers, any number can be faulty, but only a subset are group members, and correctness depends upon fewer than a third of the group members being faulty. Castro and Liskov fix group membership [24], and correctness depends upon fewer than a third of servers total being faulty. Castro and Liskov note that if too many correct servers are incorrectly removed from the group in Rampart, the remaining faulty servers may violate correctness [24, Section 8]. One benefit of a group membership model is that such protocols can perform well and ensure correctness in an environment with many faulty and non-responsive servers so long as the group invariant is maintained. To emulate the fixed group membership in Castro and Liskov's model, Rampart can simply lower the ranking of group members that would be removed (i.e., remove and rejoin suspected members).

Several protocols use unreliable failure detectors [36, 53, 54, 67], which allow for a modular protocol design. For example, the SecureRing protocol uses an unreliable failure detector to provide an asynchronous Byzantine fault-tolerant group communication abstractiong [54], which can be used to build a replicated state machine. As in Rampart, SecureRing uses a secure group membership protocol. Rampart [87] and SecureRing [54] also batch operations to minimize cryptographic and network overhead (SecureRing calls batching *packing*).

5.1.4 Additional Related Work

There has been similar progress on Byzantine fault-tolerant read/write protocols, which can be used to build robust block storage systems. Recent protocols have included PASIS [44] and AVID [18]. Most recently, Hendricks et al. [48] proposed a protocol that demonstrates $<10\%$ overhead for the large data objects that characterize high-bandwidth storage applications.

Farsite [4, 35] uses a Byzantine fault-tolerant replicated state machine to manage metadata in a distributed file system. Farsite issues leases to clients for metadata such that clients can update metadata locally, which can increase scalability. Upon conflict or timeout, leases are recalled, but updates may be lost if the client is unreachable. Leasing schemes do not provide the strong consistency guarantees expected of replicated state machines (linearizability [51]), so leasing is not acceptable for some applications. Also, choosing lease timeouts presents an additional challenge: a short timeout increases the probability that a client will miss a lease recall or renewal, but a long timeout may stall other clients needlessly in case of failure. Farsite expires leases after a few hours [35], which is acceptable only because Farsite is not designed for large-scale write sharing [13].

To scale metadata further, Farsite hosts metadata on multiple independent replicated state machines called directory groups. To ensure namespace consistency, Farsite uses a special-purpose subprotocol to support Windows-style renames across directory groups [35]. This subprotocol allows Farsite to scale, but it is inflexible and does not generalize to other operations. For example, the subprotocol cannot handle POSIX-style renames [35]. Byzantine Locking would allow Farsite and similar protocols to maintain scalability without resorting to a special-purpose protocol.

Yin et al. [107] describe an architecture in which agreement on operation order is separated from operation execution, allowing execution to occur on distinct servers. But, the replicated state machine protocol is never relieved of the task of ordering operations, and as such, it remains a bottleneck to performance.

Recent research has provided techniques for improved uncommon case performance, which would complement Zzyzx's improvement of common case performance. Clement et al. [30] improve the performance of Byzantine fault-tolerant systems under attack. Singh et al. [95] propose using a *pre-serializer* to mask conflicts in quorum-based protocols, which can improve performance in the presence of contention.

Dividing the state machine into objects, as is required to achieve the benefits of Zzyzx, has been used in many previous replicated state machine systems. (Quorum systems frequently use this technique as well, as discussed above.) For example, in their work preceding Zyzzyva, Kotla et al. [57] describe CBASE, which partitions a state machine into objects and allows concurrent execution of operations known to involve only independent state. Rodrigues et al. [88] describe BASE, which partitions a state machine into objects to allow independent recovery of (abstract views of) state across non-identical replicas.

Much progress has also been made on improving non-Byzantine fault-tolerant replicated state machine protocols, such as Paxos [64]. Chandra et al. [25] hardened a production Paxos implementation with ad-hoc consistency checks to tolerate some corruptions. Mao et al. [72] proposed Mencius, a Paxos-like protocol in which the leader rotates to improve throughput.

5.2 Definitions and System Model

This chapter makes the same assumptions about network asynchrony and the security of cryptographic primitives (e.g., MACs, signatures, and hash functions), and offers the same guarantees of liveness and correctness (linearizability), as the most closely related prior works [2, 24, 32, 55]. Zzyzx tolerates up to f Byzantine faulty servers and any number of Byzantine faulty clients, given $3f+1$ servers. As will be discussed, Zzyzx allows physical servers to take on different roles in the protocol, namely as *log servers* or state machine *replicas*. A log server and replica can be co-located on a single physical server, or each can be supported by separate physical servers. Regardless of the mapping of roles to physical servers, the presentation here assumes that there are $3f+1$ log servers, at most f of which fail, and $3f+1$ replicas, at most f of which fail.

To model the strong guarantees provided by Byzantine fault tolerance, faulty nodes are assumed to be controlled and coordinated by a malicious adversary. The adversary also controls the network, deciding if and when messages are delivered and corrupting messages at will. Cryptographic message authentication codes (MACs) and signatures are used to ensure the integrity of messages between correct nodes.

Of course, such a powerful adversary could prevent progress by refusing to deliver messages. In general, a replicated state machine may not be live in an asynchronous network environment, even if only a single benign fault might occur [40]. Zzyzx implements Byzantine Locking on top of Zyzzyva, and it provides the same liveness guarantee [55]: Zzyzx is live if the network is eventually synchronous [37], i.e., there is a fixed (though potentially unknown) delay bound and some (unknown) point in time after which all messages are delivered within that bound. In general, Byzantine Locking inherits the liveness properties of the underlying protocol. For example, an obstruction-free Byzantine Locking protocol could be built on top of Q/U [2].

As in prior protocols [2, 24, 32, 55], Zzyzx satisfies linearizability [51] from the perspective of correct clients. Linearizability requires that correct clients issue operations sequentially, leaving at most one operation outstanding at a time, as assumed in this chapter, but this requirement can be relaxed. Each operation applies to one or more *objects*, which are individual components of state within the state machine.

Two operations are *concurrent* if each operation's response does not precede the other's invocation. This chapter makes a distinction between concurrency and contention. An object experiences *contention* if distinct clients submit concurrent requests to the object or interleave requests to it (even if those requests are not concurrent). For example, an object experiences frequent contention if two clients alternate writing to it. Low contention can be characterized by long *contention-free runs* on an object, comprised of multiple operations on the object by a single client.

It is precisely such contention-free runs on objects for which Byzantine Locking is beneficial, since it provides exclusive access to those objects and enables an optimized protocol to be used to invoke operations on them. As such, it is important for performance that objects be defined so as to minimize contention for them. HQ [32] and Q/U [2] divide the state machine into objects for similar reasons. CBASE [57] divides the object space to exploit request parallelism within a single state machine. Of course, the granularity depends on the application and the workload. In the extreme, if only one client is active much of the time, the state need not be partitioned at all.

Because Byzantine Locking ensures good performance in fault- and contention-free runs, the replicated state machine protocol design can focus on other goals, such as efficiently handling faults [30] or contention [24, 55], or even in simplifying the protocol. Many replication protocols elect a server as a leader, calling it the *primary* [24, 55], *coordinator* [72], or *sequencer* [95]. For simplicity and concreteness, this chapter assumes Byzantine Locking on top of Zyzzyva, so certain activities can be relegated to the primary to simplify the protocol. Take note, however, that Byzantine Locking is not dependent on a primary-based protocol, but can build on a variety of underlying replicated state machine protocols.

5.3 Byzantine Locking and Zzyzx

This section describes Byzantine Locking and Zzyzx at a high level. A more formal treatment of Byzantine Locking, including a proof of correctness and liveness, is provided in Appendix A.

Byzantine Locking provides a client an efficient mechanism to modify replicated objects by providing the client temporary exclusive access to the object. A client that holds temporary exclusive

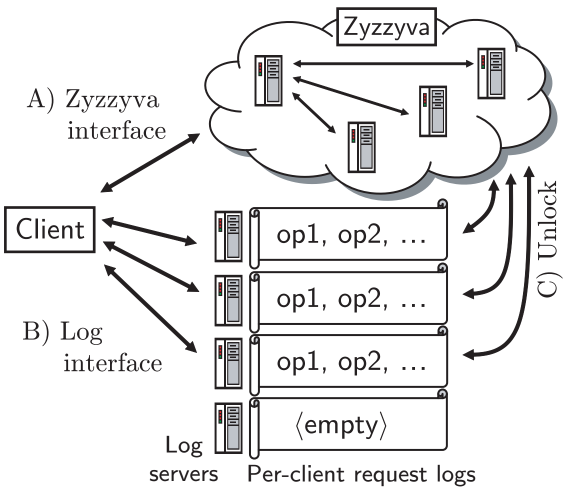


Figure 5.3.3: **Zzyzx components.** *The execution of Zzyzx can be divided into three subprotocols, described in Section 5.3.* **A)** *If a client has not locked the objects needed for an operation, the client uses a substrate protocol such as Zyzzyva (Section 5.3.1).* **B)** *If a client holds locks for all objects touched by an operation, the client uses the log interface (Section 5.3.2).* **C)** *If a client tries to access an object for which another client holds a lock, the unlock subprotocol is run (Section 5.3.3).*

access to an object is said to have *locked* the object. Zzyzx implements Byzantine Locking on top of Zyzzyva [55], as illustrated in Figure 5.3.3. In Zzyzx, objects are unlocked by default. At first, each client sends all operations through the *Zyzzyva interface* (Figure 5.3.3A). Upon realizing that there is little contention, the client sends a request through Zyzzyva to lock a set of objects. The Zyzzyva interface and the locking operation are described in Section 5.3.1.

For subsequent operations that touch only locked objects, the client uses the *log interface* (Figure 5.3.3B). The excellent performance of Zzyzx derives from the simplicity of the log interface, which is little more than a replicated append-only log. To issue a request, a client increments a sequence number and sends the request to $3f+1$ *log servers*, which may or may not be physically co-located with the Zyzzyva replicas. Each log server appends the operation to its per-client *request log* if the operation is in order, and then executes the operation on its local state before returning a response to the client. If $2f+1$ log servers provide matching responses, the operation is complete. The log interface is described further in Section 5.3.2.

If another client attempts to access a locked object through the Zyzzyva interface, the primary initiates the *unlock subprotocol* (Figure 5.3.3C). The primary sends a message to each log server to unlock the object. The log servers reach agreement on their state using the Zyzzyva interface, mark the object as unlocked, and copy the updated object back into the Zyzzyva replicas. If the client that locked the object subsequently attempts to access the object through the log interface, the log server replies with an error code, and the client retries its request through the Zyzzyva interface. The unlock subprotocol is described further in Section 5.3.3.

5.3.1 The Zyzzyva Interface and Locking

In Zzyzx, each client maintains a list of locked objects that is the client's current best guess as to which objects it has locked. The list may be inaccurate without impacting correctness. Each replica, including the primary, maintains a special state machine object called the *lock table*. The lock table provides an authoritative description of which client, if any, has currently locked each object. The lock table also provides some per-client state, including a variable vs. Each object is initially marked unlocked in the lock table, vs is set to 1 for each client, and each client's list of its locked objects is empty.

Upon invoking an operation in Zzyzx, a client checks if any object touched by the operation is not in its list of locked objects, in which case the client uses the Zyzzyva interface. As in Zyzzyva, the client sends its request to the primary replica. The primary checks if any object touched by the request is locked. If not, the primary resumes the standard Zyzzyva protocol, batching requests as needed and sending ordering messages to the other replicas as usual.

If an object touched by the request is locked, the primary initiates the unlock subprotocol, described in Section 5.3.3. The request is enqueued until all touched objects are unlocked. Any subsequent request to lock an object touched by an enqueued request is enqueued as well (or, alternatively, denied). As objects are unlocked, the primary dequeues each enqueued request for which all objects touched by the request have been unlocked, and resumes the standard Zyzzyva protocol as above.

Note that a client can participate in Zzyzx using only the Zyzzyva protocol, and in fact does not need to be aware of the locking mechanism at all. In general, a replicated state machine protocol can be upgraded to support Byzantine Locking without affecting legacy clients.

A client can attempt to lock its working set to improve its performance. To do so, the client sends a *lock request* for each object using the Zyzzyva protocol. The replicas evaluate a deterministic *locking policy* to determine whether to grant the lock. If granted, the client adds the object to its list of locked objects. The replicas also return the value of the per-client vs variable, which is incremented upon unlock. The variable vs stands for view-stamp, but this chapter will refer to it as vs to prevent confusion with the view-stamp in PBFT and Zyzzyva. The vs variable is used to synchronize state between the log servers and Zyzzyva replicas.

If there is little concurrency across a set of objects, the entire set can be locked in a single operation. For example, each file in a file system could be represented by an object, and a client's entire home directory subtree could be locked upon login, using the efficient log interface for nearly all operations.

The Zzyzx prototype uses a simple policy to decide when to lock an object: each replica counts how often a single client accesses an object without contention. The client sets a flag in its request stating that it would like to lock touched objects. If the count reaches a threshold and the flag is set, the client locks the object. (The evaluation in Section 5.5 uses a threshold of ten.) Counters are only kept for recently accessed objects. Every few thousand operations, each counter is reset, which allows any client to lock the object. Reseting the counters makes locking multiple objects as above more likely. Thus, a client will only lock an object if no other client has recently accessed the object.

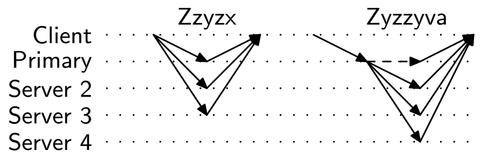


Figure 5.3.4: **Basic communication pattern of Zzyzx versus Zyzzyva.** *Operations on locked objects in Zzyzx complete in a single round-trip with* $2f+1$ *log servers. Zyzzyva requires three message delays, if all* $3f+1$ *replicas are responsive, or more message delays, if some replicas are unresponsive.*

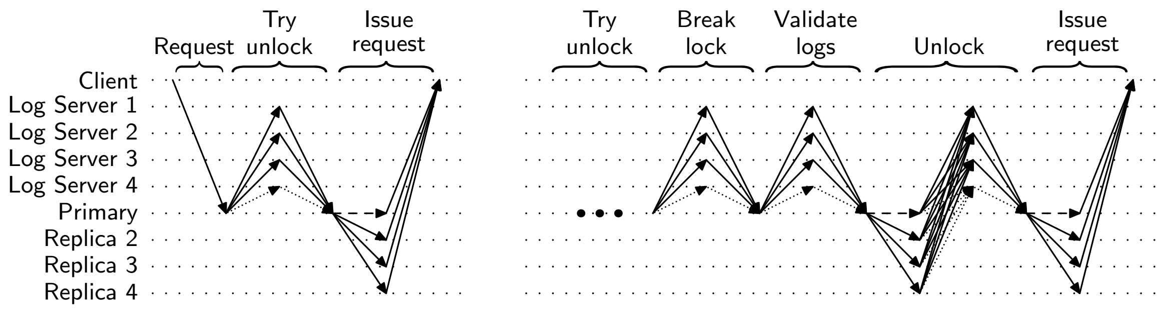
5.3.2 The Log Interface

Upon invoking an operation in Zzyzx, a client may find that all objects touched by the operation are in its list of locked objects, in which case the client uses the log interface. The client increments its request number, which is a local counter used for each operation issued through the log interface, and builds a message containing the request, the request number, and the vs. It then computes a MAC of the message for each log server, as in prior protocols. Unlike prior protocols, the client then computes another MAC of the message and the first set of MACs for each log server. The inner MACs are used in the unlock subprotocol. The client then sends the message along with the inner MACs and the appropriate outer MAC to each log server. Thus, the outer MAC is a standard authenticator. The layered set of MACs is called a *layered authenticator*.

Upon receiving a request, each log server verifies the outer MAC. The log server then verifies that the request is in order as follows: If the request number is lower than the most recent request number for the client, the request is a duplicate and is ignored. If the request number matches the most recent number, the most recent response is re-sent. If the request number is greater than the next in sequence, or if the vs value is greater than the log server's value, the log server must have missed a request so it initiates state transfer (described in Section 5.4.1). If the log server has promised not to access an object touched by the request (since the object is in the process of being unlocked, as described in Section 5.3.3), it returns failure.

If the request number is next in sequence, the log server tries to execute the request. It lazily fetches objects from replicas as needed by invoking the Zyzzyva interface. Of course, if a log server is co-located with a Zyzzyva replica, pointers to objects may be sufficient. If fetching an object fails because the object is no longer locked by the client, the log server returns failure. Otherwise, the log server has a local copy of each object that is touched by the request. It executes the request on its local copy, appends the request, vs, request number, and the inner set of MACs to its request log, and returns a response. Upon receiving $2f+1$ non-failure responses, the client returns the majority response.

If some log server returns failure, the client sends a special *retry request* through the Zyzzyva interface, which includes both the request and the request number. Each replica checks if the request completed at the log servers before the last execution of the unlock subprotocol, in which case the replicas tell the client to wait for a response from a log server. Otherwise, the replicas execute the request as a normal Zyzzyva request.



A) Fast unlock B) Full unlock

Figure 5.3.5: **Unlock message diagram. A)** In the absence of faults and concurrency, the fast unlock subprotocol is executed (Section 5.3.3). The primary fetches a hash of the request log at $2f+1$ log servers (labeled "Try Unlock"). If hashes match, the primary sends the hash values, which unlock the object, and the conflicting request through Zyzzyva in a batch ("Issue request"). **B)** Otherwise, the full unlock subprotocol is executed (Section 5.3.3). The primary fetches request logs from $2f+1$ log servers ("Break lock"). It then asks each log server to validate the inner client MACs in the request logs ("Validate logs"). Log servers agree on the longest valid request log using Zyzzyva ("Unlock"), and they replay that request log to reach a consistent state. Finally, as above, the log servers send the primary matching hashes, which the primary sends with the conflicting request through Zyzzyva ("Issue request").

Figure 5.3.4 shows the basic communication pattern of the log interface in Zzyzx versus Zyzzyva. Zyzzyva requires 50% more network hops than Zzyzx, and Zyzzyva requires all $3f+1$ servers to be responsive to perform well, f more than the $2f+1$ responsive servers that Zzyzx requires. Zzyzx improves upon Zyzzyva further, though, by removing the bottleneck primary and requiring less cryptography at servers. The latter improvement obviates the need for batching, improving latency without cost to throughput. Batching is a technique used in previous protocols [21, 55] where the primary accumulates a batch of requests before sending them to other replicas. Batching allows the cryptographic overhead of the agreement subprotocol to be amortized over many requests, but waiting for a batch of requests before execution can increase latency. Because Byzantine Locking provides clients temporary exclusive access to objects, each client can order its own requests for locked objects, avoiding the need for an agreement or even a speculative agreement [55] subprotocol.

5.3.3 Handling Contention

The protocol, as described so far, is a simple combination of operations issued to Zyzzyva (Section 5.3.1) and requests appended to a log (Section 5.3.2). The magic of Byzantine Locking is found in the unlock subprotocol, which differentiates Byzantine Locking from prior lease- and lock-like mechanisms found in systems such as Farsite [4] and Chubby [16].

A client that does not hold a lock on an object uses the Zyzzyva interface, as described in Section 5.3.1. Similarly, a client that receives a failure response from a log server retries that request through the Zyzzyva interface, as described in Section 5.3.2. Either way, the client sends its request to the primary, which checks if any objects touched by the request are locked, as described in Section 5.3.1. If an object is locked, the primary initiates the unlock subprotocol, described in this section. Though this section describes unlocking a single object, implementations can, of course, unlock multiple objects in a single execution of the unlock subprotocol. The primary is well-positioned to initiate the unlock subprotocol, because it knows which objects are locked and it controls the order in which operations are issued.

The unlock subprotocol consists of a fast path and a slower path, described below in Sections 5.3.3 and 5.3.3, respectively. Figure 5.3.5 shows the communication pattern for both types of unlock. As shown in Figure 5.3.5A, the fast unlock is quite efficient, requiring just a single round-trip between the primary and $2f+1$ log servers. The full unlock protocol, shown in Figure 5.3.5B, requires additional communication, but is required only when a client or log server is faulty, or when request logs do not match.

Fast Unlock

In the fast unlock path, shown in Figure 5.3.5A, the primary sends a "Try unlock" message to each log server, describing the object (or set of objects) being unlocked. Each log server constructs a message containing the hash of its request log and the hash of the current value of the object. A designated replier includes the value of the object in its message (as in replies for PBFT [24]). Once again, if log servers are co-located with Zyzzyva replicas, only a pointer to the object may need to be sent. Each log server sends its response to the primary formatted as a Zyzzyva request. Unlike other Zyzzyva requests, however, the log server does not wait for a response, but rather the primary resends its "Try unlock" message until enough log servers provide responses.

Upon receiving $2f+1$ responses with matching object and request log hashes and at least one object that matches the hashes, the primary sends the responses through the standard Zyzzyva protocol, batched with any requests enqueued due to object conflicts (see Section 5.3.1). Before sending a response to the primary, each log server adds the object to a list of objects it promises not to touch until the next instantiation of the full unlock subprotocol, described in Section 5.3.3 below.

Making matching logs more likely: The fast path requires that request logs match, which may not be the case if the messages for a request from the client and the messages for a "Try unlock" from the primary arrive in a different order at different log servers, such that the primary's message precedes the client's request on some log servers but vice-versa on others. Fortunately, concurrent requests are less common than in other protocols, because only two nodes are involved (a single client and the primary). The window of vulnerability for interleaving is the jitter between client requests divided by the round-trip time (the client has at most one outstanding request). This window is similar to the one in Q/U [2], and better than the one in HQ [32] (HQ's window spans two write phases). Furthermore, depending on network topology, multicast may enforce a de facto ordering, allowing concurrency only if a packet is dropped between the last switch before the log servers.

To further increase the likelihood that request logs match, each log server can send a hash for the request log up to the most recent request that touched the object (or set of objects) being unlocked. Thus, a client request concurrent with an unlock operation only forces a full unlock if an object is needed by both the request and the unlock.

The Full Unlock Subprotocol

If the request logs do not match in "Try unlock" from the fast path, the full unlock subprotocol (Figure 5.3.5B) must be initiated. The primary fetches signed request logs from $2f+1$ log servers. (Signatures can be avoided using standard techniques, but full unlock is rare, so signatures are used to simplify the protocol description.) Before sending its request log, a log server adds the object (or set of objects) being unlocked to its list of objects that it promises not to touch until the next full unlock, as in a fast unlock. The primary then sends these request logs to each log server, which validates its MAC from the inner layer of MACs stored with each request in the request log (see Section 5.3.2). Log validation can be skipped if the longest request log matches at $f+1$ log servers.

The log servers then vote on which request logs contain valid MACs by issuing a Zyzzyva request. The longest request log for which $f+1$ log servers found valid MACs is returned to each log server, which then replays the log as needed to reach a consistent state that matches the state at other correct log servers. The log server then marks the object being unlocked as unlocked, increments vs, and clears the list of objects it promised not to touch until the next full unlock. Finally, as in fast unlock in Section 5.3.3, correct log servers send the primary matching hash values describing their state and the object to be unlocked. The primary sends these hash values, along with any requests enqueued due to object conflicts (see Section 5.3.1), in a batch through the standard Zyzzyva protocol.

Figure 5.3.5B illustrates a few optimizations over the complete pseudo-code for the full unlock subprotocol. In particular, several sequential Zyzzyva requests can be issued in parallel, as shown in the "Unlock" phase in Figure 5.3.5B. The full unlock subprotocol is similar to view change in PBFT or Zyzzyva, and is not needed in fault- and concurrency-free executions.

5.4 Protocol Details

The log servers use checkpointing and state transfer mechanisms, similar to mechanisms found in PBFT [24], HQ [32], and Zyzzyva [55], described in Section 5.4.1. As in Q/U [2] and HQ [32], Zzyzx takes advantage of *preferred quorums*. Section 5.4.2 describes optimizations for read-only requests, more aggressive locking, lower contention, and preferred quorums. Zzyzx can provide near-linear scalability by deploying additional replicas, discussed in Section 5.4.3. Such scalability is unprecedented—the throughput of most Byzantine fault-tolerant protocols cannot be increased by adding additional replicas, because all requests flow through a bottleneck node (e.g., the primary in PBFT [24] and Zyzzyva [55]) or overlapping quorums (which provides limited scalability). Appendix B provides further details.

Though this paper assumes that at most f servers (log servers or replicas) fail, Byzantine Locking (and many other Byzantine fault-tolerant protocols) can support a hybrid failure model that allows for different classes of failures. As in Q/U [2], suppose that at least $n-t$ servers are correct, and that at least $n-b$ are *honest*, i.e., either correct or fail only by crashing; as such, $t \geq b$. Then, the total number of servers is $b+2t+1$ rather than $3f+1$, and the quorum size is $b+t+1$ rather than $2f+1$. Of course, when $f=b=t$, it is the case that $b+2t+1=3f+1$ and $b+t+1=2f+1$. The benefit of such a hybrid model is that one additional server can provide the benefits of Byzantine fault-tolerance. (Or, more generally, b additional servers can tolerate b simultaneous Byzantine faults.) A hybrid model suits deployments where arbitrary faults, such as faults due to soft errors, are less common than crash faults.

5.4.1 Checkpointing and State Transfer

Log servers should checkpoint their state periodically both to truncate their request logs and to limit the amount of work needed for a full unlock. The full unlock operation acts as a checkpointing mechanism, because log servers reach an agreed-upon state. Thus, upon full unlock, requests prior to the unlock can be purged. The simplest checkpoint protocol is for log servers to execute the full unlock subprotocol for a null object at fixed intervals. Zzyzx can also use standard checkpointing techniques found in Zyzzyva [56] and similar protocols, which may be more efficient.[1]

If a correct client sends a request number greater than the next request number in order, the log server must have missed a request. The log server sends a message to all $3f$ other log servers, asking for missed requests. Upon receiving matching values for the missing requests and the associated MACs from $f+1$ log servers, the log server replays the missed requests on its local state to catch up. Since $2f+1$ log servers must have responded to each of the client's previous requests, at least $f+1$ correct log servers must have these requests in their request logs. A log server may substitute a stable checkpoint in place of prior requests.

5.4.2 Optimizations

Read-only requests: A client can read objects locked by another client if all $3f+1$ log servers return the same value, as in Zyzzyva. Thus, read-only operations perform no worse. If $2f+1$ log servers return the same value and the object was not modified since the last checkpoint, the client can return that value. If the object was modified, the client can force a checkpoint, which may be less expensive than the unlock subprotocol.

Aggressive locking: If an object is locked but never fetched by log servers through the Zyzzyva interface, there is no need to run the unlock subprotocol under contention. The primary can just send the conflicting request through the standard Zyzzyva protocol, which will deny future fetch requests pertaining to the previous lock. Thus, aggressively locking a large set of objects does not lower performance.

Pre-serialization: Section 5.5.7 finds that Zzyzx outperforms Zyzzyva for contention-free runs as short as ten operations. The pre-serializer technique of Singh et al. [95] could make the break-even point even lower.

Preferred quorums: Rather than send requests to all $3f+1$ log servers for every operation, a client can send requests to $2f+1$ log servers if all $2f+1$ servers provide matching responses. This optimization limits the amount of data sent over the network, which is useful when the network is bandwidth- or packet-limited, or when the remaining f replicas are slow. It also frees f servers to

[1]Such protocols operate as follows. A log server computes a *tentative checkpoint* at fixed request intervals, which consists of the log server's current state. Each log server sends a MACed hash of its tentative checkpoint to every other log server. Upon accumulating $2f+1$ such messages that match, a log server sends a *checkpoint* message to every other log server, which includes the hash from the tentative checkpoint. Upon accumulating $2f+1$ checkpoint messages, the checkpoint is stable and prior state can be purged.

process other tasks or operations in the common case, thus allowing a factor of up to $\frac{3f+1}{2f+1}$ higher throughput.

Performance under attack: Byzantine fault-tolerant prototypes often perform poorly, if at all, in malicious environments [30]. The protocol description in Section 5.3 does not address performance under attack in the interest of clarity and simplicity, but standard techniques to detect, isolate, and mitigate faulty behavior can be applied (e.g., [30, 46], [21, Section 3.2.2]).

5.4.3 Scalability Through Log Server Groups

There is nothing in Section 5.3 that requires the group of replicas used in the Byzantine fault-tolerant replicated state machine protocol to be hosted on the same servers as the log servers. Thus, a system can deploy replicas and log servers on distinct servers. Similarly, the protocol can use multiple distinct groups of log servers. An operation that spans multiple log server groups can always be completed through the Zyzzyva interface. The benefit of multiple log server groups is near linear scalability in the number of servers, which far exceeds the scalability that can be achieved by adding servers in prior protocols.

For example, given $4f+2$ replicas and assuming cross-group operations are rare, the replicas can be divided into two log server groups with non-overlapping preferred quorums of size $2f+1$. Thus, the protocol can scale by roughly a factor of $2\times$ given an additional $f+1$ servers. The first $2f+1$ servers would comprise the preferred quorum of the first log server group, and the second $2f+1$ servers would comprise the preferred quorum of a second log server group. The state machine replicas would be hosted on any subset of $3f+1$ servers.

Because most operations would touch only half of the servers, the achievable system throughput would be double that of a typical implementation. Furthermore, this technique may allow log servers in a wide-area distributed system to be located geographically closer to clients. For example, $2f+1$ log servers could be deployed in each of the California, Massachusetts, Texas, and Washington offices of a company. Most operations would operate efficiently on local copies of the data in request logs. Operations that spanned sites could be completed through the replicated state machine protocol run over $3f+1$ replicas.

An application can benefit from multiple log server groups only if many operations execute on disjoint subsets of objects and if objects do not need to be transferred often. Though not all applications can benefit, several important applications in most need of Byzantine fault tolerance meet these requirements. For example, many distributed storage and database applications are embarrassingly parallelizable. In fact, Zzyzx was designed in the course of architecting large-scale, high-performance Byzantine fault-tolerant storage systems [1, 48]. In such a system, Zzyzx would manage metadata, and data would be accessed through a Byzantine fault-tolerant block storage protocol (e.g., [18, 44, 48]).

5.5 Evaluation

This section evaluates the performance of Zzyzx and compares it with that of Zyzzyva and that of an unreplicated server. Zzyzx is implemented as a module on top of Zyzzyva, which in turn is a modified version of the PBFT library [24]. MD5 was replaced with SHA1 in Zyzzyva and Zzyzx, because MD5 is no longer considered secure [104]. (Zyzzyva also uses AdHash, which is known to be insecure [103].) Zyzzyva is the only other Byzantine fault-tolerant protocol measured, because it outperforms prior Byzantine fault-tolerant protocols [55, 96].

Because Zyzzyva does not utilize Byzantine Locking, replicas must agree on the order of requests before a response is returned to the client (rather than the client ordering requests on locked objects). Order agreement requires that the primary generates MACs for each of the $3f+1$ other servers, which would prove expensive if done for every operation. Thus, the primary in Zyzzyva (like in PBFT) orders multiple operations at once. The primary accumulates a *batch* of size B of requests. It then orders all B operations and computes a single set of MACs, amortizing the cryptographic cost over several requests. Though important for high throughput in Zyzzyva, batching increases latency in Zyzzyva. This section considers Zyzzyva using batch sizes of one (B=1) and ten (B=10).

5.5.1 Assumptions and Limitations

Zzyzx always ensures correctness, but it was not designed to perform well in a malicious environment. There are two reasons for this design choice. First, in many environments, corruptions are relatively rare. Thus, performance only matters in the common case, when there are no corruptions. Second, the design did not consider performance in malicious environments to avoid additional complexity. Most Byzantine fault-tolerant replicated state machine protocols perform poorly in malicious environments, but some recent proposals introduce techniques that Zzyzx could adopt to ensure reasonable performance even in the presence of malicious attackers [7, 30].

Zzyzx requires that the state machine can be partitioned into objects. As discussed in Section 5.1, many other protocols impose similar requirements, and many important systems are amenable to partitioning into objects. Furthermore, the scalability of Zzyzx depends upon the ability to partition the workload into log server groups. Fortunately, as discussed in Section 5.4.3, many important workloads, such as distributed storage workloads, are easily partitionable.

5.5.2 Experimental Setup

All experiments are performed on a set of computers that each have a 3.0 GHz Intel Xeon processor, 2 gigabytes of memory, and an Intel PRO/1000 network card. All computers are connected to a HP ProCurve Switch 2848, which has a specified internal bandwidth of 96 Gbps (69.3 Mpps). Each computer runs Linux kernel 2.6.28-7 (the most recent at the time of writing) with default networking parameters. Experiments use the Zyzzyva code released by the protocol's authors [55], configured to use all optimizations [55, 56]. Both Zyzzyva and Zzyzx use UDP multicast. After accounting for the performance difference between SHA1 and MD5, the evaluation of Zyzzyva agrees with that of Kotla et al. [55].

A Zyzzyva replica process runs on each of $3f+1$ server computers. For Zzyzx, except where noted, one Zyzzyva replica process and one log server process run on each of $3f+1$ server computers. Zzyzx is measured both with the preferred quorum optimization of Section 5.4.2 enabled (labeled "Zzyzx ") and with preferred quorums disabled (labeled "Zzyzx-noPQ").

The micro-benchmark workload used in Sections 5.5.3 through 5.5.5 consists of each client process performing a null request and receiving a null reply. This workload is the same as the

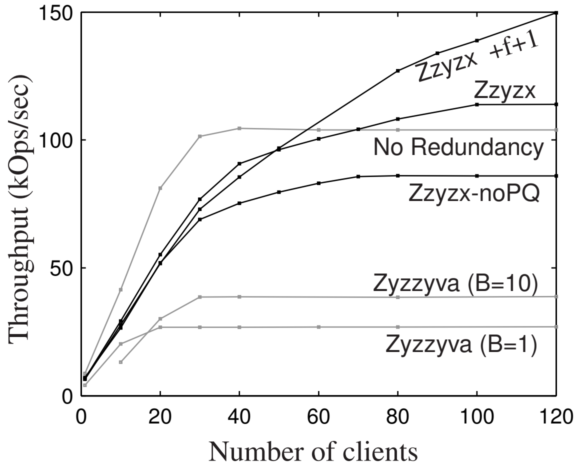


Figure 5.5.6: **Throughput vs. client processes when $f = 1$ and all servers are responsive.**

workload measured by Kotla et al. [55], which is based on the 0/0 micro-benchmark of Castro and Liskov [24]. Each client accesses an independent set of objects, avoiding contention. A client running Zzyzx locks each object on first access. The workload is meant to highlight the overhead found in each protocol, as well as to provide a basis for comparison by reproducing prior experiments.

Each physical client computer runs 10 instances of the client process. This number was chosen so that the client computer does not become processor-bound. All experiments are run for 90 seconds, with measurements taken from the middle 60 seconds. The mean of at least 3 runs is reported, and the standard deviation for all results is within 3% of the mean.

5.5.3 Scalability

Figure 5.0.1 on page 58 shows the throughput of Zzyzx and Zyzzyva as the number of servers increases when tolerating one fault. The first $3f+1$ log servers are co-located with the Zyzzyva replicas. Additional log servers run on dedicated computers. Unlike prior protocols, including Zyzzyva, the performance of Zzyzx improves as more servers are utilized. Since only $2f+1$ log servers are involved in each operation and independent log server sets do not need to overlap, the increase in usable quorums results in nearly linear scalability. Zyzzyva and previous protocols do not support scaling in this fashion. Although data is only shown for $f = 1$, the general shape of the curve applies when tolerating more faults.

5.5.4 Throughput

Figure 5.5.6 shows the throughput achieved, while varying the number of clients, when tolerating a single fault and when all servers are correct and responsive. Throughput is not noted for B=10 when using fewer than 10 clients. Zzyzx significantly outperforms all Zyzzyva configurations. Zzyzx's maximum throughput is 2.9× that of Zyzzyva with B=10, and higher still compared to Zyzzyva without batching. When Zzyzx is run on f+1 additional servers (6 total), it's maximum throughput

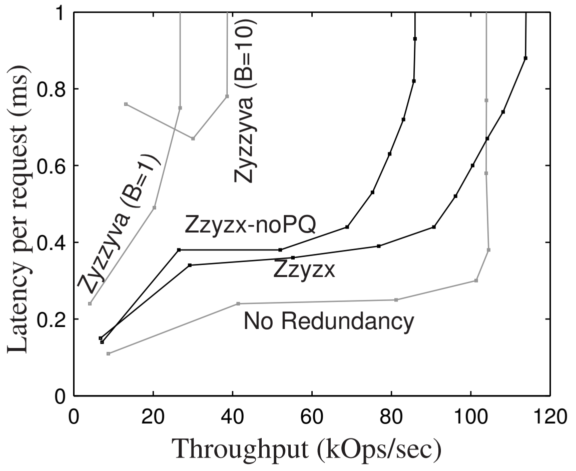


Figure 5.5.7: **Latency vs. throughput when $f = 1$ and all servers are responsive.**

is 3.9× that of Zyzzyva with B=10. Even without preferred quorums, Zzyzx's maximum throughput is 2.2× that of Zyzzyva with B=10, due to Zzyzx's lower network and cryptographic overhead.

Due to the preferred quorums optimization, Zzyzx provides higher maximum throughput than the unreplicated server, which simply generates and verifies a single MAC, because each log server processes only a fraction ($\frac{2f+1}{3f+1}$) of the requests. With preferred quorums disabled (ZZYZX-NOPQ), Zzyzx provides lower throughput than the unreplicated server due to checkpoint, request log, and network overheads.

5.5.5 Latency

Figure 5.5.7 shows the average latency for a single operation as the applied load is varied. When serving a single request, Zzyzx exhibits 39–43% lower latency than Zyzzyva, and Zzyzx continues to provide lower latency as load increases. The lower latency is because Zzyzx requires only 2 one-way message delays (33% fewer than Zyzzyva), each server computes fewer MACs, and log servers in Zzyzx never wait for a batch of requests to accumulate before executing the request and returning a response.

Figure 5.5.7 also illustrates the problem with batching in Zyzzyva. Unless replicas are saturated, Zyzzyva provides lower latency when batching is disabled, but batching allows substantially higher maximum throughput.[2] Whereas batching forces a choice between low latency and a high throughput in Zyzzyva, Zzyzx can provide both low latency and high throughput.

[2]PBFT uses feedback from subsequent protocol phases to tune the batch window, avoiding the latency increase seen in Zyzzyva. Zyzzyva eliminates subsequent protocol phases from PBFT and so cannot use this technique. But, Zyzzyva still exhibits lower latency than PBFT because requests complete after fewer message delays and require less cryptography.

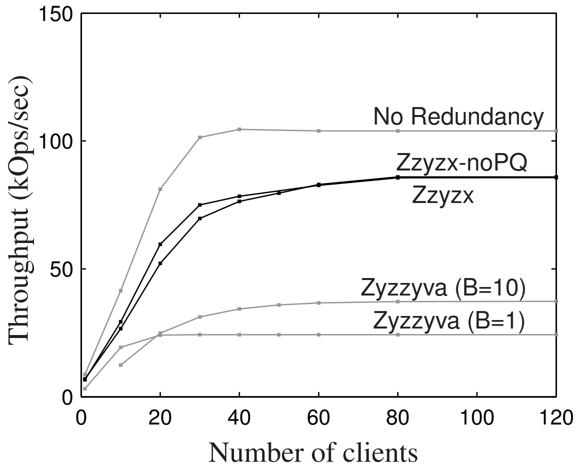


Figure 5.5.8: **Throughput vs. client processes when $f = 1$ and one server is unresponsive.**

5.5.6 Performance with f Slow Servers

Figure 5.5.8 shows the throughput achieved while varying the number of clients when configured to tolerate a single fault and when a single server is unresponsive though not malicious (e.g., crashed). The performance of an unreplicated server is included for comparison.

Zzyzx throughput decreases approximately 33%, to the unaffected level of Zzyzx-noPQ, because all requests now use the same $2f+1$ servers. Once clients detect that a server in a preferred quorum of the log server group is unresponsive, they send requests to all servers in the log server group and stay on the fast path. With or without preferred quorums, when one server is unresponsive, Zzyzx provides 55% higher throughput than Zyzzyva.

Figure 5.5.8 for Zyzzyva with B=10 reports substantially better throughput than reported by Kotla et al. at SOSP [55], because the released Zyzzyva code uses the "commit optimization" described in their extended technical report [56]. The commit optimization, however, requires an extra message delay in the form of an all-to-all round of communication. Due to this all-to-all communication, Zyzzyva with the commit optimization may not be as "fault scalable" [2] as the Zyzzyva protocol reported by Kotla et al. at SOSP [55].

5.5.7 Performance Under Contention

Figure 5.5.9 shows the performance of Zzyzx under contention. For this workload, each client accesses an object a fixed number of times before the object is unlocked. The client then procures a new lock and resumes accessing the object. The experiment is designed to identify the *break-even* point of Zzyzx, which is the length of the shortest contention-free run for which Zzyzx outperforms Zyzzyva.

When batching in Zyzzyva is disabled to improve latency, Zzyzx outperforms Zyzzyva for contention-free runs that average 10 or more operations. Zzyzx outperforms Zyzzyva when batching is enabled (B=10) for contention-free runs that average 20 or more operations. Zzyzx achieves

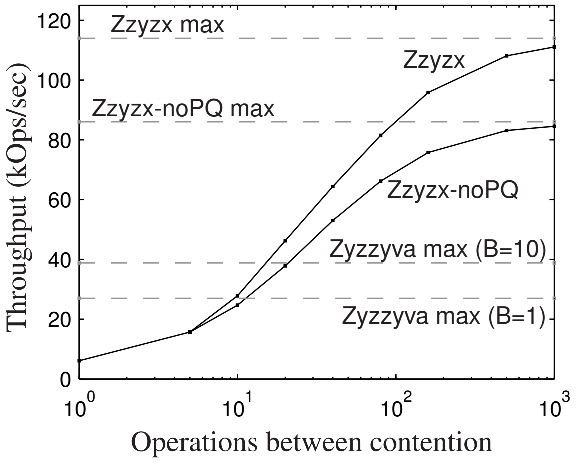


Figure 5.5.9: **Throughput vs. number of consecutive contention-free ops when $f = 1$ and all servers are responsive. The horizontal lines show the throughput of Zyzzyva and Zzyzx in the absence of contention.**

85–90% of its maximum throughput for contention-free runs of 160 operations—as noted in Section 5.1.1, contention-free runs often average in the thousands.

5.5.8 Postmark and Trace-driven Execution

Figure 5.5.10 compares the execution of Zzyzx and Zyzzyva on a file system workload. Zzyzx outperforms prior replicated state machines for both data and metadata due to its minimal communication, but custom block transport protocols can use erasure coding to outperform replicated state machines [44, 48]. Thus, this evaluation focuses on difference in the performance between protocols when managing the metadata of a distributed file system.

To test a metadata workload, a memory-backed file system using FUSE was implemented that interfaces with Zyzzyva, Zzyzx, and an unreplicated server for metadata operations. Zzyzx completed 60% more transactions per second (TPS) than Zyzzyva in the default Postmark benchmark [52], where a transaction may consist of multiple Zzyzx or Zyzzyva operations plus some processing time. Zzyzx completes 22% fewer transactions per second than the unreplicated server. Postmark produces a workload with many small files, similar to the workload found in a mail server. Postmark performance depends primarily upon request response time.

Metadata operations were extracted from NFS traces of a large departmental server. Over 14 million metadata operations were considered from the Harvard EECS workload between Monday 17–Friday 21 of February 2003 [38]. A matching workload mix was then executed on Zyzzyva and Zzyzx. Zzyzx used the lock policy described in Section 5.3.1 to determine when to lock an object.

The operations in the trace were 55% read-only, for which Zyzzyva used its one round-trip read optimization. Because read-only operations in Zyzzyva avoid the primary, they perform similarly to the Zzyzx-noPQ line. Zzyzx used the log interface for 82% of operations, with an av-

	Postmark transactions	Trace-based throughput
Zzyzx	590 TPS	97.7 kOps/sec
Zyzzyva	369 TPS	51.0 kOps/sec
Unreplicated	757 TPS	

Figure 5.5.10: **Performance of Zzyzx and Zyzzyva for a file system workload.**

erage contention-free run length of 4926 operations. Of the 18% of operations executed through the Zyzzyva interface, 56% were read-only and used Zyzzyva's one round-trip read optimization. Overall, Zzyzx provided a factor of 1.6× higher throughput than Zyzzyva.

5.6 Conclusion

Byzantine Locking allows creation of efficient and scalable Byzantine fault-tolerant services. Compared to the state-of-the-art (Zyzzyva), Zzyzx delivers a factor of 2.2×–2.9× higher throughput during concurrency-free and fault-free operation, given the minimum number of servers ($3f+1$). Moreover, unlike previous Byzantine fault-tolerant replicated state machine protocols, Zzyzx offers near-linear scaling of throughput as additional servers are added.

Chapter 6

Conclusion

This dissertation makes a number of contributions to the field of Byzantine fault-tolerant distributed system design, both in overall protocol design and in techniques that should be useful in other future protocols. It offers the following three conclusions and a few suggestions for future work:

Byzantine fault-tolerant erasure-coded storage systems can provide similar latency and throughput as systems that tolerate only crashes: This dissertation demonstrates that Byzantine fault-tolerant erasure-coded storage systems can be implemented using similar hardware resources as systems that tolerate only crashes (only $m+2f \geq 3f+1$ servers) without introducing much computational overhead beyond a checksum of the data or much network overhead beyond an additional roundtrip when writing large blocks of data.

Byzantine fault-tolerant replicated state machines can service requests in a single roundtrip to *t+b+1* responsive servers with minimal cryptographic overhead: Applications such as a distributed metadata service require the richer semantics of a state machine. Prior Byzantine fault-tolerant replicated state machine protocols either required that $3f+1$ or more nodes participate in each request or provided lower throughput and higher latency. Zzyzx provides higher throughput and lower latency than prior protocols, and only $2f+1$ servers need participate in a request in the common case. More precisely, when tolerating b Byzantine faulty servers and t crash faults ($t \geq b$), Zzyzx requires $t+b+1$ servers to participate in each request, out of $2t+b+1$ total servers. Thus, when $b=1$, Zzyzx can provide many of the benefits of Byzantine fault tolerance with just a single additional server. To achieve this result, Zzyzx requires that the workload exhibit low object contention. Fortunately, many important application, such metadata services, experience low contention.

Byzantine fault-tolerant replicated state machines can scale through workload partitioning: Block storage protocols can scale by distributing blocks of data across a larger set of servers, but more general services such as a metadata cluster face more challenging scalability. Many prior protocols could not scale by partitioning the workload because requests were ordered through a single, centralized server. Zzyzx provides scalable fault tolerance through Byzantine Locking, which allows the workload of a distributed system to be partitioned across distinct sets of If an operation interacts with state partitioned across different servers, the state is aggregated such that the operation remains atomic. Thus, in addition to providing higher throughput and lower latency than prior Byzantine fault-tolerant replicated state machines, Zzyzx introduces unprecedented scalability.

Future work: This dissertation could be extended in a number of ways. Homomorphic fingerprinting uses Reed-Solomon codes [85], Rabin's Information Dispersal Algorithm [84], and other linear erasure coding schemes. Some recent erasure coding schemes are more efficient (e.g.. [81]), but may generate fewer fragments, and may not be linear and hence not compatible with homomorphic fingerprinting. A potential extension could consider efficient linear erasure coding schemes that generate many fragments.

Byzantine fault-tolerant replicated state machine protocols are perceived as complicated, but most share common features. For example, clients can either access servers directly, or clients can send messages through a pre-serializer [95] or primary. The direct approach improves latency but suffers under contention (e.g., Kursawe [63], Q/U [2], HQ [32], and Byzantine Locking). The pre-serializer approach increases latency, but avoids contention (e.g., PBFT [24] and Zyzzyva [55]). Singh et al. considered adding a pre-serializer to direct-access protocols and found similarities to primary-based protocols [95]. A potential extension could continue to modularize such concepts.

Appendix A

The Correctness of Byzantine Locking

This appendix provides a proof of the safety and liveness properties of Byzantine Locking and Zzyzx. Zzyzx can be divided into two types of objects, the *manager object* and the *log object*. The manager object manages lock state and processes requests on objects that experience high concurrency. It is implemented using a Byzantine fault-tolerant replicated state machine protocol such as Zyzzyva [55] or PBFT [24]. There is a log object for each client, and each log object processes requests for objects that have been locked by its client. Zzyzx achieves better performance through an optimized implementation of the log object.

Section A.1 provides specifications of object types. Section A.1.1 defines the object-based state machine, which describes the type of applications that Zzyzx can efficiently implement. Zzyzx can implement state machines similar to those considered by PBFT [24], Q/U [2], HQ [32], and Zyzzyva [55]. As in some of these protocols, Zzyzx divides application state into objects to ensure efficiency. Section A.1.2 and Section A.1.3 provide the sequential specification of the log object and the manager object. Section A.2 describes how a manager object and a set of log objects can be combined to form a linearizable Byzantine fault-tolerant replicated state machine. Section A.3 describes how a client would submit requests in such a system and demonstrates that such a system is live.

A.1 Sequential Specifications of Relevant Objects

This section provides the sequential specification of the object-based state machine, the log object, and the manager object. The term "object" is used in more than one context. In the distributed systems community, a distributed protocol is used to implement an object, such as a log object, manager object, or state machine object. But, in a different context, the implementation of a state machine is often broken into small chunks of state called objects [2, 24, 32, 55], either as a practical matter or for protocol reasons. This chapter will differentiate between the two by always referring to the log object, the manager object, or the state machine object in full. Other references to the word object correspond to portions of state for a state machine implementation.

A.1.1 The Object-Based State Machine Object

An object-based state machine object consists of object state for a bounded number of objects and the $\overline{\textbf{execute\_request}}$ operation. Each object is referred to by its object identifier, denoted oid, and the value of an object is denoted $\mathsf{state}[\mathsf{oid}]$. Each $\mathsf{state}[\mathsf{oid}]$ is initialized to a default value. The $\overline{\textbf{execute\_request}}$ operation takes as input the current state and a request, applies the request to the current state, and returns the new state and a response. In summary, $\langle \mathsf{response}, \mathsf{state}' \rangle \leftarrow \overline{\textbf{execute\_request}}(\mathsf{req}, \mathsf{state})$.

An identifier for the client that submitted the request is often included in the request. For notational simplicity, the pseudo-code will use the function **execute_request**, which differs from $\overline{\textbf{execute\_request}}$ in two ways. First, the client identifier cid is written explicitly. Second, **execute_request** modifies state, so it need not return state' (i.e., state is passed by reference). Thus, in the pseudo-code, the state machine is called as follows: $\mathsf{response} \leftarrow \textbf{execute\_request}(\mathsf{cid}, \mathsf{req}, \mathsf{state})$, which is equivalent to $\langle \mathsf{response}, \mathsf{state} \rangle \leftarrow \overline{\textbf{execute\_request}}(\mathsf{req}, \mathsf{state})$ if req includes cid.

Byzantine Locking requires a compact representation of the set of objects that a request may touch, denoted $\textbf{oids\_touched}(\mathsf{req})$, and an efficient mechanism to check whether an object may be touched by a request before executing the request, denoted $\mathsf{oid} \in \textbf{oids\_touched}(\mathsf{req})$. The application of requests is deterministic. Thus, the application of requests on disjoint sets of objects commutes. That is, the relative order of requests that touch non-overlapping sets of objects does not matter. For example, suppose

$$\langle \mathsf{response}_1, \mathsf{state}_1 \rangle \leftarrow \overline{\textbf{execute\_request}}(\mathsf{req}_1, \mathsf{state}_0)$$
$$\langle \mathsf{response}_2, \mathsf{state}_2 \rangle \leftarrow \overline{\textbf{execute\_request}}(\mathsf{req}_2, \mathsf{state}_1)$$

and

$$\langle \mathsf{response}'_2, \mathsf{state}'_1 \rangle \leftarrow \overline{\textbf{execute\_request}}(\mathsf{req}_2, \mathsf{state}_0)$$
$$\langle \mathsf{response}'_1, \mathsf{state}'_2 \rangle \leftarrow \overline{\textbf{execute\_request}}(\mathsf{req}_1, \mathsf{state}'_1)$$

If req_1 and req_2 operate on disjoint sets of objects (that is, $\textbf{oids\_touched}(\mathsf{req}_1) \cap \textbf{oids\_touched}(\mathsf{req}_2) = \emptyset$), then the order of the requests does not matter, and so $\mathsf{response}_1 = \mathsf{response}'_1$, $\mathsf{response}_2 = \mathsf{response}'_2$, and $\mathsf{state}_2 = \mathsf{state}'_2$.

Note the difference between state machine interfaces that require objects to be specified in advance (as in Zzyzx, HQ [32], and Q/U [2]) and interfaces that do not (as in Zyzzyva [55] and PBFT [24]). That is, some state machines do not require an efficient and compact function $\textbf{oids\_touched}(\mathsf{req})$ that can be used to specify which objects are touched in advance. In practice, specifying a working set of objects in advance is easy in many important systems, such as file systems. Furthermore, state machine implementations often already divide the state into objects to allow efficient checkpointing and state transfer. But, interfaces that do not require advance descriptions of objects touched can easily support some applications that are challenging to support in an efficient manner using interfaces that require advance descriptions (e.g., consider dereferencing a pointer within an object). Of course, $\textbf{oids\_touched}(\ldots)$ can be conservative, perhaps even reporting that all objects may be touched for each operation, so both interfaces can implement the same set of applications, though an overly conservative implementation of $\textbf{oids\_touched}(\ldots)$ may impact the performance of Zzyzx, HQ [32], and Q/U [2].

A.1.2 The Log Object

A *log object* consists of object state, denoted by the array $\mathsf{state}[]$; the request log and its position, denoted $\mathsf{request\_log}$ and reqno; and a nondecreasing value vs for synchronization with the manager object. Initially, $\mathsf{state}[*] = \text{NULL}$, $\mathsf{request\_log}[*] = \text{NULL}$, $\mathsf{reqno} = 0$, and $\mathsf{vs} = 1$. A *log object* supports three operations: APPEND, IMPORT, and FETCH+UNLOCK. If $\mathsf{state}[\mathsf{oid}] = \text{NULL}$, then oid is said to be unlocked; otherwise, oid is locked. Operations are associated with message tags for remote procedure calls in the pseudo-code, and so each request is written out as $\langle \text{TAG}, \mathit{args}\rangle$. The sequential specification of each operation is as follows:

$\langle$**APPEND**$, \mathsf{cid}, \mathsf{req}\rangle$ If each object touched by req is locked, then let $\mathsf{reqno} \leftarrow \mathsf{reqno} + 1$, $\mathsf{request\_log}[\mathsf{reqno}] \leftarrow \mathsf{req}$, and perform **execute_request**$(\mathsf{cid}, \mathsf{req}, \mathsf{state})$ and return its result. Otherwise, return FAILURE.

$\langle$**IMPORT**$, \mathsf{vs}', \mathsf{oid}, \mathsf{obj}\rangle$ If $\mathsf{vs}' = \mathsf{vs}$ and $\mathsf{state}[\mathsf{oid}] = \text{NULL}$, then let $\mathsf{state}[\mathsf{oid}] \leftarrow \mathsf{obj}$.

$\langle$**FETCH+UNLOCK**$, \mathsf{oid}\rangle$ Let $\mathsf{state}[\mathsf{oid}] \leftarrow \text{NULL}$, let $\mathsf{vs} \leftarrow \mathsf{vs} + 1$, and return the triplet $\langle \mathsf{vs}, \mathsf{oid}, \mathsf{request\_log}\rangle$.

In a typical execution, a client will IMPORT several objects and APPEND several requests. If FAILURE is returned, the client retries the request at the manager object. If object oid is needed at the manager object, a client can invoke FETCH+UNLOCK and replay $\mathsf{request\_log}$ at the manager object, which unlocks oid. In Zzyzx, each log object is associated with a single designated client (denoted cid) that invokes operations APPEND and IMPORT, such that the argument to APPEND is always cid.

A.1.3 The Manager Object

A *manager object* consists of object state, denoted by the array $\mathsf{state}[]$; nondecreasing values $\mathsf{vs}_{\mathsf{cid}}$ for synchronizing with the log object for designated client cid; an array $\mathsf{locktable}[]$, such that $\mathsf{locktable}[\mathsf{oid}] = \text{NULL}$ if oid is unlocked or $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$ if oid is locked by the log object for designated client cid; and $\mathsf{reqno}_{\mathsf{cid}}$ describing the most recently replayed request for cid. It supports three operations: EXEC, LOCK, and REPLAY+UNLOCK. Initially, $\mathsf{state}[\mathsf{oid}]$ is set to a default value for each oid, $\mathsf{vs}_{\mathsf{cid}} = 1$, $\mathsf{locktable}[*] = \text{NULL}$, and $\mathsf{reqno}_{\mathsf{cid}} = 0$. A client identifier cid uniquely identifies the designated client. For any designated client, there is at most one log object. A client can be the designated client for multiple log objects by using multiple client identifiers. Variable $\mathsf{reqno}_{\mathsf{cid}}$ counts the requests replayed for client cid, not all executed requests. The sequential specification of each operation is as follows:

$\langle$**EXEC**$, \mathsf{cid}, \mathsf{req}\rangle$ If each oid touched by req is unlocked ($\mathsf{locktable}[\mathsf{oid}] = \text{NULL}$), then perform **execute_request**$(\mathsf{cid}, \mathsf{req}, \mathsf{state})$ and return its result. Otherwise, return FAILURE.

$\langle$**LOCK**$, \mathsf{cid}, \mathsf{oid}\rangle$ If $\mathsf{locktable}[\mathsf{oid}] = \text{NULL}$, then $\mathsf{locktable}[\mathsf{oid}] \leftarrow \mathsf{cid}$ and return $\langle \mathsf{vs}_{\mathsf{cid}}, \mathsf{oid}, \mathsf{state}[\mathsf{oid}]\rangle$. Otherwise, return FAILURE.

$\langle$**REPLAY+UNLOCK**$, \mathsf{cid}, \mathsf{vs}', \mathsf{oid}, \mathsf{request\_log}\rangle$ If $\mathsf{vs}' < \mathsf{vs}_{\mathsf{cid}} + 1$, return FAILURE. Otherwise, do as follows. First, let $\mathsf{vs}_{\mathsf{cid}} \leftarrow \mathsf{vs}'$. Second, while $\mathsf{request\_log}[\mathsf{reqno}_{\mathsf{cid}} + 1] \neq$ NULL let $\mathsf{reqno}_{\mathsf{cid}} \leftarrow \mathsf{reqno}_{\mathsf{cid}} + 1$ and perform **execute_request**$(\mathsf{cid}, \mathsf{request\_log}[\mathsf{reqno}_{\mathsf{cid}}], \mathsf{state})$. Third, if $\mathsf{oid} \neq$ NULL and $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$, set $\mathsf{locktable}[\mathsf{oid}] \leftarrow$ NULL. Fourth, return SUCCESS.

In a typical execution, a client invokes EXEC and may LOCK its working set of objects. To allow access to an object locked by a log object, a client invokes FETCH+UNLOCK at the log object before invoking REPLAY+UNLOCK to reconcile state between the log object and the manager object, which unlocks the object.

A.2 Linearizability

This section describes how a manager object and a set of log objects can be combined to form a linearizable Byzantine fault-tolerant replicated state machine.

A.2.1 The Reads-from Relation

DEFINITION A.2.1. A REPLAY+UNLOCK operation *reads from* a FETCH+UNLOCK operation if the cid passed to REPLAY+UNLOCK is the designated cid for the log object at which FETCH+UNLOCK was invoked and if FETCH+UNLOCK returns the $\langle \mathsf{vs}, \mathsf{oid}, \mathsf{request\_log}\rangle$ triplet that matches the arguments to REPLAY+UNLOCK.

DEFINITION A.2.2. An IMPORT operation *reads from* a LOCK operation if the cid passed to LOCK is the designated cid for the log object at which IMPORT was invoked and the $\langle \mathsf{vs}, \mathsf{oid}, \mathsf{obj}\rangle$ triplet returned by LOCK matches the arguments to IMPORT.

DEFINITION A.2.3. An operation history of a manager object and a set of log objects is *reads-from valid* if each REPLAY+UNLOCK reads from a FETCH+UNLOCK operation and each IMPORT reads from a LOCK operation.

A.2.2 The Equivalence of Replayed Requests

In this section, consider a reads-from valid operation history. Consider the call to **execute_request**$(\ldots)$ during a REPLAY+UNLOCK for some $\mathsf{reqno}_{\mathsf{cid}}$ such that $\mathsf{req} = \mathsf{request\_log}[\mathsf{reqno}_{\mathsf{cid}}]$:

$$\mathsf{retval} \leftarrow \textbf{execute\_request}(\mathsf{cid}, \mathsf{req}, \mathsf{state}) \tag{A.2.4}$$

Consider the call to **execute_request**$(\ldots)$ during an APPEND at the log object with designated client cid for some reqno' such that $\mathsf{req}' = \mathsf{request\_log}[\mathsf{reqno}']$:

$$\mathsf{retval}' \leftarrow \textbf{execute\_request}(\mathsf{cid}, \mathsf{req}', \mathsf{state}') \tag{A.2.5}$$

This section will show that the $\mathsf{retval} \leftarrow \textbf{execute\_request}(\mathsf{cid},\mathsf{req},\mathsf{state})$ for $\mathsf{reqno}_{\mathsf{cid}}$ in Equation A.2.4 at the manager object during a REPLAY+UNLOCK corresponds to some $\mathsf{retval}' \leftarrow \textbf{execute\_request}(\mathsf{cid},\mathsf{req}',\mathsf{state}')$ for reqno' in Equation A.2.5 at the log object during an APPEND. Lemma A.2.6 shows that $\mathsf{req} = \mathsf{req}'$ and $\mathsf{reqno}_{\mathsf{cid}} = \mathsf{reqno}'$. Lemma A.2.8 shows that $\mathsf{retval} = \mathsf{retval}'$, and that, after $\textbf{execute\_request}(\ldots)$ at the manager object and log object, for each oid such that $\mathsf{state}'[\mathsf{oid}] \neq \text{NULL}$, it is the case that $\mathsf{state}[\mathsf{oid}] = \mathsf{state}'[\mathsf{oid}]$ and $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$.

LEMMA A.2.6. For each call to $\textbf{execute\_request}(\mathsf{cid},\mathsf{req},\mathsf{state})$ in Equation A.2.4 for $\mathsf{reqno}_{\mathsf{cid}}$ by operation $\langle \text{REPLAY+UNLOCK}, \mathsf{cid}, *, *, * \rangle$ at the manager object, there exists a call to $\textbf{execute\_request}(\mathsf{cid},\mathsf{req}',\mathsf{state}')$ in Equation A.2.5 by operation $\langle \text{APPEND}, \mathsf{cid}, \mathsf{req}' \rangle$ for reqno' at the log object, such that $\mathsf{req} = \mathsf{req}'$ and $\mathsf{reqno}' = \mathsf{reqno}_{\mathsf{cid}}$.

Proof. Because the operation history is reads-from valid, the REPLAY+UNLOCK reads from a FETCH+UNLOCK that returned request_log such that $\mathsf{request\_log}[\mathsf{reqno}_{\mathsf{cid}}] = \mathsf{req}$. Because only APPEND appends requests to the request log, an APPEND for reqno' must have appended req' to request_log such that $\mathsf{reqno}' = \mathsf{reqno}_{\mathsf{cid}}$ and $\mathsf{req}' = \mathsf{req}$. □

Lemma A.2.7 will form the base case of the induction in Lemma A.2.8. In particular, Lemma A.2.7 shows that, after $\textbf{execute\_request}(\ldots)$ at the manager object and log object, for each oid such that $\mathsf{state}'[\mathsf{oid}] \neq \text{NULL}$, it is the case that $\mathsf{state}[\mathsf{oid}] = \mathsf{state}'[\mathsf{oid}]$ and $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$, but only when the last operation to modify oid at the log object was an IMPORT. Lemma A.2.8 removes the restriction that the last operation to modify oid was an IMPORT.

LEMMA A.2.7. Let $\mathsf{state}'[\mathsf{oid}]$ be the value of oid at the log object before the call to $\textbf{execute\_request}(\ldots)$ by an APPEND in Equation A.2.5 for reqno'. Suppose $\mathsf{state}'[\mathsf{oid}] \neq \text{NULL}$ and the last operation to modify oid at the log object was $\langle \text{IMPORT}, \mathsf{vs}, \mathsf{oid}, \mathsf{obj} \rangle$. Then, it is the case that $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$ and $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$, where $\mathsf{state}[\mathsf{oid}]$ and $\mathsf{locktable}[\mathsf{oid}]$ are the values at the manager object either at the end of the operation history if reqno' is not replayed, or before the call to $\textbf{execute\_request}(\ldots)$ in Equation A.2.4 for $\mathsf{reqno}_{\mathsf{cid}} = \mathsf{reqno}'$ if reqno' is replayed.

Proof. Because the operation history is reads-from valid, IMPORT reads from a LOCK that sets $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$ and returns $\langle \mathsf{vs}, \mathsf{oid}, \mathsf{obj} \rangle$ such that $\mathsf{obj} = \mathsf{state}[\mathsf{oid}]$. IMPORT then sets $\mathsf{state}'[\mathsf{oid}] \leftarrow \mathsf{obj}$, which is not modified until the APPEND (the IMPORT was the last operation to modify oid). The remainder of this proof shows that oid is not modified at the manager object after the LOCK until $\mathsf{reqno}_{\mathsf{cid}}$ is replayed, if ever, so $\mathsf{state}[\mathsf{oid}] = \mathsf{obj}$ and thus $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$. There are two cases: it must be shown that, after the lock but before $\mathsf{reqno}_{\mathsf{cid}}$ is replayed, no request in an EXEC operation and no replayed request modifies oid.

No EXEC modifies oid after the LOCK until $\mathsf{reqno}_{\mathsf{cid}}$ is replayed because oid remains locked in that period, as follows. Only FETCH+UNLOCK can set $\mathsf{state}'[\mathsf{oid}] = \text{NULL}$. Because $\mathsf{state}'[\mathsf{oid}] \neq \text{NULL}$ before the APPEND, any such FETCH+UNLOCK either precedes the IMPORT, such that the value vs' before FETCH+UNLOCK is less than vs (less than because FETCH+UNLOCK increments vs'), or follows the APPEND. A REPLAY+UNLOCK for vs' must precede a LOCK for vs if $\mathsf{vs}' < \mathsf{vs}$, and a REPLAY+UNLOCK that reads from a FETCH+UNLOCK that follows the APPEND for $\mathsf{reqno}' =$

$\mathsf{reqno}_{\mathsf{cid}}$ will replay $\mathsf{reqno}_{\mathsf{cid}}$ if $\mathsf{reqno}_{\mathsf{cid}}$ has yet to be replayed. Thus, after the LOCK but before any replay of $\mathsf{reqno}_{\mathsf{cid}}$, it is the case that $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$, and thus no EXEC modifies oid.

Similarly, no replayed request modifies oid after the LOCK until $\mathsf{reqno}_{\mathsf{cid}}$ is replayed, because the APPEND is the first APPEND to modify oid after the IMPORT. Any request replayed before $\mathsf{reqno}_{\mathsf{cid}}$ must precede not only APPEND but also IMPORT (otherwise, the last operation to modify oid would not be the IMPORT). Furthermore, because $\mathsf{state}[\mathsf{oid}] = \text{NULL}$ before the IMPORT, any prior request that modified oid must have preceded a FETCH+UNLOCK$'$ that incremented $\mathsf{vs}' < \mathsf{vs}$ and set $\mathsf{state}[\mathsf{oid}] = \text{NULL}$. Because $\mathsf{state}[\mathsf{oid}] \neq \text{NULL}$ when the prior request modifies oid, the prior request follows an $\langle \text{IMPORT}', \mathsf{vs}'', \mathsf{oid}, \mathsf{obj}\rangle$, where $\mathsf{vs}'' \leq \mathsf{vs}' < \mathsf{vs}$. Because IMPORT$'$ reads from a LOCK$'$ that returns vs'' (by reads-from valid) and because $\mathsf{vs}'' < \mathsf{vs}$, LOCK$'$ precedes LOCK. Because $\mathsf{locktable}[\mathsf{oid}] = \text{NULL}$ when LOCK succeeds, a REPLAY+UNLOCK$'$ that reads from a FETCH+UNLOCK$'$ that follows the prior request must have executed, replaying the prior request if it has yet to be replayed. Thus any request that modifies oid before $\mathsf{reqno}_{\mathsf{cid}}$ is replayed would be replayed before the LOCK for vs. □

The following lemma ties the state of the log object together with that of the manager object before and after the execution of each request.

LEMMA A.2.8. For each call to $\mathsf{retval} \leftarrow \textbf{execute\_request}(\mathsf{cid}, \mathsf{req}, \mathsf{state})$ in Equation A.2.4 for $\mathsf{reqno}_{\mathsf{cid}}$ at the manager object, there exists a call to $\mathsf{retval}' \leftarrow \textbf{execute\_request}(\mathsf{cid}, \mathsf{req}', \mathsf{state}')$ in Equation A.2.5 for reqno' at the log object such that $\mathsf{reqno}' = \mathsf{reqno}_{\mathsf{cid}}$ and $\mathsf{retval} = \mathsf{retval}'$. Furthermore, for each oid such that $\mathsf{state}'[\mathsf{oid}] \neq \text{NULL}$ during the call at the log object, during the call at the manager object it is the case that $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$, and after both calls it is the case that $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$.

Proof. That each $\textbf{execute\_request}(\ldots)$ at the manager object during a REPLAY+UNLOCK is matched by one at the log object is proven in Lemma A.2.6. The remainder of the lemma is proven by induction over the value of reqno:

Inductive Hypothesis: Consider the call to $\textbf{execute\_request}(*, *, \mathsf{state}')$ at the log object for reqno' by an APPEND in Equation A.2.5. Consider the corresponding call to $\textbf{execute\_request}(*, *, \mathsf{state})$ at the manager object for $\mathsf{reqno}_{\mathsf{cid}}$ in Equation A.2.4. For each oid such that $\mathsf{state}'[\mathsf{oid}] \neq \text{NULL}$ during the call at the log object, it is the case that $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$ during the call at the manager object and that $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$ before the calls at both the log object and the manager object.

Corollary to Inductive Hypothesis: Suppose the inductive hypothesis holds. APPEND fails if any modified object is unlocked, so, for each modified object, it is the case that $\mathsf{state}'[\mathsf{oid}] \neq \text{NULL}$ and thus $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$. Hence, because $\mathsf{req}' = \mathsf{req}$ (by Lemma A.2.6) and by determinism, it is the case that $\mathsf{retval} = \mathsf{retval}'$ and that $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$ after the calls at the log object and the manager object, as required by the lemma.

Base Case: Consider the first call to $\textbf{execute\_request}(\mathsf{cid}, *, *)$ in a REPLAY+UNLOCK at the manager object, which sets $\mathsf{reqno}_{\mathsf{cid}} = 1$, and the corresponding first call to $\textbf{execute\_request}(\mathsf{cid}, *, *)$ in the first successful APPEND at the log object, which sets $\mathsf{reqno}' = 1$. For each oid such that $\mathsf{state}'[\mathsf{oid}] \neq \text{NULL}$, the last operation to modify oid must have been an IMPORT. Thus, by Lemma A.2.7, it is the case that $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$ and $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$.

Inductive Step: Consider the call to **execute_request**$(*, *, \mathsf{state}')$ at the log object by an APPEND in Equation A.2.5, which sets reqno', and the call to **execute_request**$(*, *, \mathsf{state})$ at the manager object in Equation A.2.4, which sets $\mathsf{reqno}_{\mathsf{cid}} = \mathsf{reqno}'$. Consider each oid such that $\mathsf{state}'[\mathsf{oid}] \neq$ NULL during the call at the log object. The last operation to modify oid at the log object was either an IMPORT or an APPEND$'$. If the last such operation was an IMPORT, then, by Lemma A.2.7, it is the case that $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$ and $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$.

If the last operation to modify oid at the log object was an APPEND$'$ for some reqno'', then $\mathsf{reqno}'' < \mathsf{reqno}' = \mathsf{reqno}_{\mathsf{cid}}$ (because APPEND$'$ precedes APPEND). Because the manager object replays requests in sequence from 1 to $\mathsf{reqno}_{\mathsf{cid}}$, there is a corresponding **execute_request**$(\ldots)$ at the manager object that replays reqno''. By the inductive hypothesis at $\mathsf{reqno}'' < \mathsf{reqno}' = \mathsf{reqno}_{\mathsf{cid}}$, and its corollary, after the corresponding **execute_request**$(\ldots)$ replays reqno'' at the manager object, $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$ and $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$. Because no other APPEND modifies oid between reqno'' and reqno', no request that modifies oid is replayed between reqno'' and reqno'. Other than replayed requests, only EXEC operations modify objects. Because $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$, an EXEC will not modify oid unless oid is unlocked. Thus, if $\mathsf{locktable}[\mathsf{oid}]$ remains cid at the manager object, then $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$ before the calls to **execute_request**$(\ldots)$ for reqno', and the lemma is proven.

Suppose oid is unlocked by REPLAY+UNLOCK$'$ at the manager object after reqno'' is replayed but before reqno' is replayed. Thus, REPLAY+UNLOCK$'$ must have read from some FETCH+UNLOCK$'$ at the log object that unlocked oid and followed APPEND$'$ but preceded APPEND. This is a contradiction, because either oid will be unlocked for APPEND and APPEND will fail, or an IMPORT will lock oid, in which case the previous operation was not APPEND$'$. □

A.2.3 Requests that are not Replayed

A client will invoke APPEND operations at the log object, and each successful APPEND will be appended to the request log in order at positions $\{1, \ldots, \mathsf{reqno}\}$. The manager object replays some of these requests in order, requests $\{1, \ldots, \mathsf{reqno}_{\mathsf{cid}}\}$. Lemma A.2.8 shows that for each $\mathsf{reqno}'_{\mathsf{cid}}$ replayed by the manager object, there exists reqno' executed by the log object such that $\mathsf{reqno}' = \mathsf{reqno}'_{\mathsf{cid}}$ and the request and return values match. But, it may be the case that $\mathsf{reqno}_{\mathsf{cid}} < \mathsf{reqno}$, in which case one or more APPEND operations execute requests $\{\mathsf{reqno}_{\mathsf{cid}} + 1, \ldots, \mathsf{reqno}\}$ at the log object that are never replayed by the manager object. This section shows that the return values from such requests match the return values that would have been generated had the requests been replayed at the manager object. This fact should come as no surprise, because, if the requests were replayed in the future, their return values must match.

LEMMA A.2.9. Consider any operation history of a manager object and a set of log objects that is reads-from valid. Consider the sequence of requests in calls to **execute_request**$(\ldots)$ by the EXEC and REPLAY+UNLOCK operations at the manager object, and consider the sequence of requests in calls to **execute_request**$(\ldots)$ by the successful APPEND operations at each log object that are not replayed at the manager object. Construct a request history by concatenating the sequence of requests from the manager object followed by the sequence from each log object (in any order), and placing each return value after its request. This request history forms a legal sequential history for an object-based state machine object.

Proof. Section A.1.1 defines an object-based state machine object by the **execute_request**$(\ldots)$ function and the current object state, such that a sequential history is legal if the sequence of requests provided to **execute_request**$(*, *, \mathsf{state})$ would produce the matching return values. Thus, the sequence of requests and return values found in calls to **execute_request**$(*, *, \mathsf{state})$ by EXEC and REPLAY+UNLOCK operations at the manager object forms a legal sequential history for an object-based state machine object.

If a request that is not replayed modifies oid at the log object with designated client cid, then $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$ and $\mathsf{state}[\mathsf{oid}] = \mathsf{state}'[\mathsf{oid}]$ at the manager object at the end of the operation history, such that $\mathsf{state}'[\mathsf{oid}]$ is the value at the log object before the first request that is not replayed modifies oid, as follows. Consider the first request that is not replayed that modifies oid. If the prior request to modify oid was an IMPORT, then, by Lemma A.2.7, it is the case that $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$ and $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$. If the prior request to modify oid was an APPEND, then, by Lemma A.2.8, it is the case that $\mathsf{state}'[\mathsf{oid}] = \mathsf{state}[\mathsf{oid}]$ and $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$.

Because $\mathsf{state}[\mathsf{oid}] = \mathsf{obj} = \mathsf{state}'[\mathsf{oid}]$, and because the return value of **execute_request**$(\ldots)$ depends only on the request and the current object state, any sequence of requests and return values from each log object can be appended to the sequence of requests from the manager object while remaining a legal history. Because $\mathsf{locktable}[\mathsf{oid}] = \mathsf{cid}$, the sequences from each log object operate on disjoint sets of objects, and thus can be appended in any order (see Section A.1.1) while remaining a legal sequential history. □

A.2.4 Real-time precedence: Reads-from Strict

For any history, operation o is said to precede operation o' if the response to operation o precedes the invocation of operation o' in the history. Thus, a history imposes a partial ordering on its operations according to this precedence relationship. The precedence relationship of a history of events from the execution of a distributed system is called real-time precedence.

REMARK A.2.10. The partial order imposed by a history can be represented as a directed acyclic graph (a DAG on operations). Consider the operation history of a manager object and a set of log objects. Choose a linearization of each log object and the manager. The linearization provides additional edges to the DAG such that the operations on each object are totally ordered. The DAG remains acyclic because linearization respects the real-time precedence reflected in the operation history.

DEFINITION A.2.11. A *reads-from edge* is an edge $\langle \mathsf{o}, \mathsf{o}' \rangle$ from operation o to operation o' such that operation o' reads from operation o.

DEFINITION A.2.12. An operation history is *reads-from strict* if, for each reads-from edge $\langle \mathsf{o}, \mathsf{o}' \rangle$, operation o precedes operation o' in the operation history.

REMARK A.2.13. If operation o precedes operation o', then there is an edge from o to o' in the operation history. Thus, the DAG of a reads-from strict operation history includes each reads-from edge. (If edge $\langle \mathsf{o}, \mathsf{o}' \rangle$ is a reads-from edge, then edge $\langle \mathsf{o}, \mathsf{o}' \rangle$ is in the DAG.)

A.2.5 The Linearizability of EXEC and APPEND

THEOREM A.2.14. Consider any operation history of a manager object and a set of log objects that is reads-from valid and reads-from strict. The request history consisting of the requests and return values from each EXEC and successful APPEND operation in the operation history is linearizable.

Proof. Lemma A.2.9 proves that the request history consisting of the requests and return values from each EXEC and REPLAY+UNLOCK operation concatenated with the requests and return values by successful APPEND operations that are not replayed forms a legal sequential history. Lemma A.2.8 proves that each request and return value in a REPLAY+UNLOCK operation is matched by a request and return value in an successful APPEND operation. Thus, there is a request history consisting of the requests and return values from each EXEC and APPEND operation in the operation history that is a legal sequential history. The remainder of this proof, then, must prove that there is such a history that respects real-time precedence.

Consider the legal sequential history from Lemma A.2.9, formed by concatenating EXEC and REPLAY+UNLOCK operations and appending un-replayed APPEND operations. Replace each REPLAY+UNLOCK by the corresponding sequence of APPEND operations that are replayed to form a sequence of all of the successful EXEC and APPEND operations. Note that the subsequence of EXEC operations at the manager object and the subsequence of APPEND operations for each log object are in the order executed by the manager object and each log object. Furthermore, note that each APPEND operation can be moved earlier, so long as it follows the LOCK of each object that it touches and remains in the same order relative to other APPENDs at that log object.

This order is precisely the order constraint imposed by the DAG in Remark A.2.10 if the operation history is reads-from strict: the EXEC and APPEND operations are in the relative order imposed by the linearization of the manager object and each log object, each APPEND follows the IMPORT that follows the LOCK for each object that it modifies, and each APPEND precedes the FETCH+UNLOCK that precedes the REPLAY+UNLOCK that replays the APPEND. Thus, total ordering of the DAG in Remark A.2.10 for a reads-from valid and reads-from strict operation history produces a linearization of the request history consisting of the requests and return values from each EXEC and APPEND operation. □

Thus, to build a linearizable replicated state machine from a manager object and a set of log objects, each client should issue a request and return its result through an EXEC operation or a successful APPEND operation. If an APPEND operation fails at a log object, the request can be retried in an EXEC operation at the manager object. If an EXEC operation fails at the manager object, the locked objects that are causing the failure can be unlocked and the EXEC operation can be retried. The next section considers the implementation of the client so as to ensure liveness.

A.3 The Client and The Primary

For any reads-from valid and reads-from strict operation history of a manager object and a set of log objects, Section A.2 shows that the request history consisting of the requests and return values found in the successful EXEC and APPEND operations is linearizable. This section describes how

```
pid.primary(op):
2000: if (op = ⟨EXEC, cid, req⟩) then
2001:     for (oid ∈ oids_touched(req) : locktable[oid] ≠ NULL) do
2002:         cid ← locktable[oid]
2003:         ⟨vs′, oid′, request_log⟩ ← logobj_cid(⟨FETCH+UNLOCK, oid⟩)
2004:         mgr_order(⟨REPLAY+UNLOCK, cid, vs′, oid′, request_log⟩)
2005:
2006: if (op ≠ ⟨REPLAY+UNLOCK, ∗, ∗, ∗, ∗⟩) then
2007:     mgr_order(op)
```

Figure A.3.1: **Primary pseudo-code.** A correct primary unlocks objects that are touched by an EXEC operation before the EXEC operation is ordered.

a client would operate in such a system as to ensure liveness as well as the reads-from valid and reads-from strict properties.

A.3.1 Reads-from Valid and Reads-from Strict

Reads-from valid and reads-from strict properties can be ensured as follows. To invoke an IMPORT or a REPLAY+UNLOCK, a client must first provide proof in the invocation of IMPORT or REPLAY+UNLOCK that the LOCK or FETCH+UNLOCK from which the IMPORT or REPLAY+UNLOCK reads has completed. Thus, each REPLAY+UNLOCK reads from a FETCH+UNLOCK operation and each IMPORT reads from a LOCK operation, satisfying reads-from valid (Definition A.2.3). Furthermore, consider each reads-from edge from a LOCK to an IMPORT or from a FETCH+UNLOCK to a REPLAY+UNLOCK. Because the LOCK or FETCH+UNLOCK completes before the IMPORT or REPLAY+UNLOCK is invoked, the LOCK or FETCH+UNLOCK precedes the IMPORT or REPLAY+UNLOCK in the operation history, satisfying reads-from strict (Definition A.2.12).

In Zzyzx, the manager object that executes the LOCK or the log object that executed the FETCH+UNLOCK provide cryptographic proof of their completion to the client. The client must provide this cryptographic proof before an IMPORT or REPLAY+UNLOCK can be invoked that reads from the LOCK or FETCH+UNLOCK. Even if the client is faulty, the existence of the proof of the response to the LOCK or FETCH+UNLOCK in the invocation of IMPORT or REPLAY+UNLOCK ensures that the response preceded the invocation.

A.3.2 The Primary

Some replicated state machine protocols elect a temporary designated leader, known as the *primary*, from the set of servers, known as *replicas*. All operations are then ordered through the primary, which allows efficient operation under contention. If the primary orders an operation correctly, the protocol ensures that the operation completes. Because a faulty primary may not properly order operations, such protocols must ensure either that the primary orders each operation, such that the operation completes, or that a new primary is elected.

Such protocols elect a new primary based on timeouts, as follows. If a client does not get a response quickly enough, it broadcasts the operation to all replicas. The replicas forward the operation to the primary and start a timer. If the timer runs out before the operation is ordered, each

```
$LockedOids \leftarrow \emptyset$        /* Initialize client's list of locked objects */
cid.issue(req):                          /* Issue a request */
2100: for (oid : /*oid meets lock criteria*/) do
2101:     retval ← mgrobj(⟨LOCK, cid, oid⟩)
2102:     if (retval ≠ FAILURE) then
2103:         ⟨vs, oid, obj⟩ ← retval
2104:         $\mathbf{logobj}_{\mathsf{cid}}$(⟨IMPORT, vs, oid, obj⟩)
2105:         $LockedOids \leftarrow LockedOids \cup \{\mathsf{oid}\}$
2106: end for
2107: if (∄ oid ∈ oids_touched(req) : oid ∉ LockedOids) then
2108:     retval ← $\mathbf{logobj}_{\mathsf{cid}}$(⟨APPEND, cid, req⟩)
2109:     if (retval ≠ FAILURE) then
2110:         return retval
2111: end if
2112:
2113: /* Couldn't use logobj, so use mgrobj */
2114: $LockedOids \leftarrow LockedOids \setminus$ oids_touched(req)
2115: return mgrobj(⟨EXEC, cid, req⟩)
```

Figure A.3.2: **Client pseudo-code.** The client tries to lock its working set of objects. If it believes it has locked all objects that are touched, it invokes APPEND at the log object. If APPEND fails or a touched object is locked, it invokes EXEC at the manager object.

replica votes to elect a new primary. After enough votes, a new primary is chosen (often in round-robin fashion among the replicas). For each new primary, the timeout value is increased (often by a small multiplicative factor). It must be the case that, given an infinite number of elections, a correct primary is elected an infinite number of times (e.g., by choosing among $f+1$ or more replicas in a round-robin fashion).

Zzyzx uses the primary at the manager object to invoke FETCH+UNLOCK and REPLAY+UNLOCK operations. Because the primary orders all operations, it can ensure that an EXEC operation never returns FAILURE, as shown in Figure A.3.1. When a client invokes an operation on the manager object (e.g., **mgrobj**(op = ⟨EXEC, req⟩)), the protocol provides the operation to the primary (pid.**primary**(op) in Figure A.3.1), which must order the operation (**mgr_order**(op) on line 2007). Upon receiving an ⟨EXEC, cid, req⟩ operation that touches one or more locked oids (line 2000), each oid is unlocked as follows. For each oid that is touched and locked (line 2001), a correct primary invokes a FETCH+UNLOCK (line 2003) followed by a matching REPLAY+UNLOCK (line 2004) to unlock the object. Only the primary invokes REPLAY+UNLOCK (line 2006), so REPLAY+UNLOCK will succeed. The FETCH+UNLOCK is invoked at the log object with the designated client that locked the oid (line 2002). Once any touched objects are unlocked, the EXEC operation is ordered (line 2007).

Thus, if the primary is correct, EXEC never returns FAILURE. To ensure these semantics, replicas that implement the manager object reject EXEC operations that touch locked objects (not shown), because such messages must be from a faulty primary. If not ordered correctly, the EXEC operation will time out, and a new primary will be elected, until eventually the EXEC operation is ordered properly such that it completes. Hence, either the current primary will order each operation correctly, or a new primary will be elected until the operation is ordered.

Figure A.3.1 describes the unlocking process as progressing sequentially (lines 2001–2004), but implementations should unlock multiple objects concurrently both by invoking FETCH+UNLOCK for log objects with different designated clients, and for modifying FETCH+UNLOCK to unlock multiple oids at once. Furthermore, there is no reason to block REPLAY+UNLOCK operations, EXEC operations that touch only unlocked oids, or LOCK operations that do not lock oids touched by a pending EXEC.

A.3.3 The Client

The client implementation is shown in Figure A.3.2. The client tries to lock whatever objects it believes it may find useful (lines 2100–2106), perhaps based on recent usage. If locking succeeds at the manager object (line 2101), the client imports the object into its log object (line 2104) and updates its list of locked objects (line 2105). It is important to note that the client's list of locked objects is its best guess as to which objects are locked, but the primary may unlock objects without notifying the client. If the client believes each object touched by the request is locked (line 2107), it invokes APPEND at the log object (line 2108). If APPEND succeeds, the client returns the response (line 2110). If the client has not locked an object that is touched by a request, or if APPEND fails, the client invokes EXEC at the manager object (line 2115). Because the manager object must unlock all objects touched by the request, the client marks all touched objects as unlocked (line 2114).

In Figure A.3.2, the request history for Zzyzx clients consists of requests and return values from EXEC operations (line 2115) and successful APPEND operations (line 2110), as required by Theorem A.2.14, which ensures that Zzyzx is linearizable. Also, locking objects is shown as an explicit action (line 2101), but it could just as well be implicit. If a client accesses an object many times in a row without other clients accessing the same object, the manager could lock object for the client by default. Similarly, objects can be locked in batches rather than one at a time. And, of course, a more compact representation of the set of objects than a list could be used (e.g., "all files under the client's home directory except those under subdirectory temp").

A.3.4 Liveness

Because a consensus protocol cannot ensure liveness in an asynchronous environment [40], liveness must depend on assumptions about synchrony. To ensure eventual progress, primary-based protocols typically assume that, after some period of time, message delays are bounded. If message delays are bounded, then the time a correct primary needs to order operations will be bounded, as well, because it is proportional to message delay. Suppose an operation does not complete. Then, a new primary will be elected and the timeout increased. Given infinitely many elections, a correct primary will be elected an infinite number of times, and the timeout will approach infinity. But, this is a contradiction, because the time needed by a correct primary to order operations is bounded and will thus eventually be less than the timeout, allowing the correct primary to order the operation. In short, liveness depends upon the timeout eventually increasing to a larger value than the time needed to order operations.

More formally, let Δ be a bound on message delay (accounting for needed retransmissions), let t represent elapsed time, let δ be the amount of time a correct primary needs to order an operation (δ is proportional to Δ), and let $\mathsf{timeout}$ be the current value of the timeout. A common synchrony assumption is *partial synchrony* [37], which states that, for possibly unknown values of Δ and τ, after time τ, message delay is bound by Δ. If $\delta < \mathsf{timeout}$, a correct primary will be able to order operations such that they complete. If $\delta < \mathsf{timeout}$ continues to hold when $\mathsf{t} > \tau$ for some time τ, then any correct primary can order operations after time τ. So long as the $\mathsf{timeout}$ increases and a new primary is elected when an operation is not ordered, then, under partial synchrony, it is the case that each operation will be ordered correctly or that a correct primary will eventually be elected with $\mathsf{timeout} > \delta$ and $\mathsf{t} > \tau$, such that the correct primary will order the operation. Either way, each operation will be ordered such that it completes, ensuring liveness.

Let ε be a bound on the response time for any invocation on the log object after time τ.[1] Suppose the primary for the manager object invokes FETCH+UNLOCK on log objects bounded by ε at most k times before ordering an operation (where k is bounded for a bounded number of objects). Then, if $\varepsilon \cdot \mathsf{k} + \delta < \mathsf{timeout}$, the primary will not time out and the operation will be ordered such that it completes. Because $\mathsf{timeout}$ increases when a new primary is elected, it will be the case that either operations are being ordered correctly, avoiding new elections, or that $\mathsf{timeout}$ will increase until $\varepsilon \cdot \mathsf{k} + \delta < \mathsf{timeout}$, such that the next correct primary can order operations, ensuring liveness. The following lemma states this fact more formally.

LEMMA A.3.1. The manager object, modified such that the primary orders operations and invokes FETCH+UNLOCK and REPLAY+UNLOCK as to prevent EXEC from returning FAILURE, remains live under partial synchrony.

Proof. Because the primary is responsible for invoking REPLAY+UNLOCK, each client invokes only EXEC and LOCK operations. If an operation is a LOCK operation or if it is an EXEC operation that does not touch any locked objects, then a correct primary can order the operation in time bounded by δ. If the EXEC operation touches one or more locked objects, a correct primary must unlock each locked object by invoking FETCH+UNLOCK. Each FETCH+UNLOCK will complete in time ε. Because there are a bounded number of objects, FETCH+UNLOCK is invoked a bounded number of times, less than some constant k. Thus, a correct primary will be able to order the operation in time bounded by $\varepsilon \cdot \mathsf{k} + \delta$.

If an operation is not ordered, replicas will time out, forcing the election of a new primary and increasing the $\mathsf{timeout}$. Because $\mathsf{timeout}$ increases upon leader election, but Δ is fixed but unknown for partial synchrony, and because ε and δ are proportional to Δ, the sum $\varepsilon \cdot \mathsf{k} + \delta$ is bounded and will thus be less than $\mathsf{timeout}$ after enough elections. Given infinitely many elections, a correct primary is elected infinitely many times. Thus, either operations will be ordered by the current primary such that they complete, or after enough elections, a correct primary will be elected and $\varepsilon \cdot \mathsf{k} + \delta < \mathsf{timeout}$ such that the correct primary can order each operation. □

The liveness theorem follows immediately from the preceding lemma.

THEOREM A.3.2. The Zzyzx protocol is live, such that each request issued by a correct client is executed and returns a return value.

Proof. Consider the request flow in Figure A.3.2. The client calls **logobj**(...) (line 2104 and line 2108), which is implemented by some replicated state machine protocol that is live by assumption. The client also calls **mgrobj**(...) (line 2101 and line 2115), which is live by Lemma A.3.1. Both are called a finite number of times, ensuring that the Zzyzx protocol is live. □

[1]Note that completing an operation in PBFT or Zyzzyva when the primary is correct requires a small, bounded amount of computation and a few message delays, taking time proportional to Δ. A faulty primary will order the operation before $\mathsf{timeout}$ or a new primary will be elected at most a fixed number of times before a correct node is primary. Thus, the time required to complete an operation on a log object if implemented by such a protocol can be bounded by some ε.

A.3.5 Obstruction-Free Variants

Zzyzx is built on top of PBFT or Zyzzyva, so it is natural to ensure similar liveness semantics. But, it is worth mentioning that Byzantine Locking has a natural implementation in an obstruction-free [50] framework such as Q/U [2]. As in Zzyzx, each client issues EXEC, APPEND, LOCK, and possibly IMPORT operations. But, instead of the primary handling FETCH+UNLOCK and REPLAY+UNLOCK, each client issues such commands to unlock objects. Thus, a client may find itself repeatedly unlocking an object in the face of interference from another client, but, as required under obstruction freedom, a client can complete any request when there is no such interference.

Appendix B

Zzyzx Optimizations

The log object operations are constructed as to enable the optimized implementation in this section and in Sections 5.3 through 5.4. This chapter provides details to bridge the implementation from Chapter 5 with the formal model from Appendix A.

B.1 Faulty Client Isolation

Since each log object accepts requests from a single designated client, Byzantine locking provides isolation between faulty clients. If a client is faulty, the REPLAY+UNLOCK operation need only ensure that a correct client could have issued the operations found in the request log. To do so, REPLAY+UNLOCK stops replaying requests upon encountering a request that touches an object that the client has not locked. Isolating faulty clients provides significant design flexibility with which to implement the log object. For example, if the client is faulty, FETCH+UNLOCK can return an arbitrary request log, and APPEND and IMPORT need not even always return a response.

B.2 Separating UNLOCK from FETCH

This section splits the FETCH+UNLOCK operation into three operations, UNLOCK, FETCH, and NEXT_VS, which enables the optimizations in Section B.3. To support UNLOCK, FETCH, and NEXT_VS, the log object keeps two additional data structures: an unlocking list, which is a list of oids, and log, which is the request_log that FETCH returns for the current vs. If $\mathsf{oid} \in \mathsf{unlocking}$, IMPORT cannot import oid, and any APPEND that touches oid returns FAILURE. Initially, $\mathsf{unlocking} = \emptyset$ and $\mathsf{log} = \text{NULL}$. The sequential specification of each operation is as follows:

$\langle$**UNLOCK**$, \mathsf{oid}\rangle$ Add oid to unlocking.

$\langle$**FETCH**$, \mathsf{oid}\rangle$ If $\mathsf{log} \neq \text{NULL}$, return log. If $\mathsf{log} = \text{NULL}$ and $\mathsf{oid} \in \mathsf{unlocking}$, let $\mathsf{log} \leftarrow \langle \mathsf{vs}, \mathsf{oid}, \mathsf{request\_log}\rangle$ and return log. Otherwise, return FAILURE.

$\langle$**NEXT_VS**$, \mathsf{vs}', \mathsf{oid}\rangle$ If $\mathsf{vs}' = \mathsf{vs}+1$, let $\mathsf{vs} \leftarrow \mathsf{vs}+1$, $\mathsf{state}[\mathsf{oid}] \leftarrow \text{NULL}$, $\mathsf{unlocking} \leftarrow \emptyset$, and $\mathsf{log} \leftarrow \text{NULL}$.

```
lsid.**logobj**$_{\mathsf{cid}}$($\langle$APPEND, vs$'$, reqno$'$, req$\rangle_{\sigma_{\mathsf{cid}}}$):
2300: if (vs$' >$ vs) then get_view(vs$'$)   /* Implicit NEXT_VS */
2301: if (reqno$'$ = reqno) then return response
2302: if (reqno$' >$ reqno$+1$) then missed_reqs(reqno$+1$)
2303: if (reqno$' \neq$ reqno$+1$) then return
2304: if ((oids_touched(req) $\cap$ unlocking) $\neq \emptyset$) then
2305:     return FAILURE
2306: else                  /* Execute, append to log, and return */
2307:     return try_request(req)
```

```
lsid.logobj$_{\mathsf{cid}}$($\langle$UNLOCK, vs$'$, oid$\rangle$):
2400: if (vs$' >$ vs) then get_view(vs$'$)   /* Implicit NEXT_VS */
2401:
2402: /* Process the UNLOCK operation */
2403: unlocking $\leftarrow$ unlocking $\cup$ {oid}
2404:
2405: /* Implicit FETCH */
2406: mgrobj($\langle$FETCH, cid, vs, oid, request_log$\rangle$)
```

Figure B.3.1: **Log server pseudo-code.**

To unlock an object, the primary issues UNLOCK followed by FETCH at the log object, then REPLAY+UNLOCK at the manager object. The primary issues NEXT_VS to clear log and the unlocking list before unlocking the next object.[1] Because FETCH does not increment vs, the manager object ensures during $\langle$REPLAY+UNLOCK, cid, vs$'$, $*$, $*\rangle$ that $\mathsf{vs}' = \mathsf{vs}_{\mathsf{cid}}$, and it sets $\mathsf{vs}_{\mathsf{cid}} \leftarrow \mathsf{vs}_{\mathsf{cid}} + 1$ (rather than ensuring that $\mathsf{vs}' \geq \mathsf{vs}_{\mathsf{cid}}$ and setting $\mathsf{vs}_{\mathsf{cid}} \leftarrow \mathsf{vs}'$, as in Section A.1.3).

B.3 APPEND, UNLOCK, and FETCH

In Zzyzx, a collection of $3f+1$ *log servers* implement a log object that tolerates f faults. The primary goal of the optimizations in this chapter is to enable APPEND operations to complete in a single round-trip (two one-way message delays) to $2f+1$ log servers, which improves upon prior results: Zyzzyva requires three one-way message delays to $3f+1$ replicas, and PBFT requires four one-way message delays to $2f+1$ replicas. The basic technique is as follows. A client will append its request to a request log at each of $2f+1$ log servers. To unlock an oid, the primary will tell each of $2f+1$ log servers to add oid to the unlocking list. To fetch, the primary will query $2f+1$ log servers for the most recent request log.

Because a successful APPEND will append req to request_log at $2f+1$ log servers, a subsequent FETCH at any $2f+1$ log servers will overlap on at least one correct log server, which will include req in its request_log. As described below, the manager object will choose the longest of $2f+1$ request logs, which will include req. Because UNLOCK for oid will complete at $2f+1$ log servers, a subsequent APPEND at $2f+1$ log servers that touches oid will overlap on at least one correct log server, which will return FAILURE.[2] If two or more log servers receive UNLOCK and APPEND messages in a different order, the primary will use the manager object to agree upon a canonical request log that log servers can replay to reach a consistent state.

Figure B.3.1 provides log server pseudo-code for the APPEND and UNLOCK operations. To invoke APPEND, a client sends a signed request to the log servers (the signature is denoted σ_{cid} in Figure B.3.1).[3] Each correct log server verifies that the request is next in sequence (lines 2301–

[1] Recall from Section A.3 that Zzyzx uses the primary to invoke FETCH+UNLOCK and REPLAY+UNLOCK. In other systems, clients could issue UNLOCK, FETCH, and NEXT_VS directly, without impacting correctness.

[2] Note that an UNLOCK may precede an APPEND that precedes a FETCH, such that the APPEND returns FAILURE but req is found in request_log. This operation order would not be possible without logically separating UNLOCK from FETCH in Section B.2.

[3] To handle requests that are dropped or corrupted by the network, the client periodically retransmits a request until it gets a response.

2303; see Section B.6.1 for an explanation of **missed_reqs**$(\ldots)$). If the request is next in sequence, the log server verifies that each object touched by the request is locked (not shown) and not on the unlocking list (line 2304). If not, FAILURE is returned (line 2305). Otherwise, the signed request is appended to request_log and reqno is incremented (not shown). The request is then executed and the response is returned (line 2307). Upon receiving $2f+1$ successful responses, the client returns the majority response.

The primary invokes the UNLOCK operation by sending a request to all log servers, which add the oid being unlocked to the unlocking list (line 2403). UNLOCK completes when $2f+1$ log servers respond. The primary could separately fetch the request_log from $2f+1$ log servers, but, instead, FETCH is issued concurrently with UNLOCK, ensuring that $\mathsf{oid} \in \mathsf{unlocking}$. To ensure that subsequent invocations of FETCH return the same value log for a particular vs, each log server calls the manager object directly with its request_log (line 2406).[4] The manager object uses its replicated state machine protocol to choose a canonical $\log_{\mathsf{cid}}$, as follows. Upon seeing $2f+1$ request logs from distinct log servers for cid, $\mathsf{vs}_{\mathsf{cid}}$, and matching oid, the manager object chooses the longest such request_log with correctly signed requests and sets $\log_{\mathsf{cid}} \leftarrow \langle \mathsf{oid}, \mathsf{request\_log} \rangle$.[5] Upon setting $\log_{\mathsf{cid}}$, the manager object can immediately execute $\langle \textsc{Replay+Unlock}, \mathsf{cid}, \mathsf{vs}, \mathsf{oid}, \mathsf{request\_log} \rangle$.

B.4 IMPORT

Rather than performing IMPORT explicitly, the IMPORT operation can be implicit by allowing the log object to lazily fetch objects that it is missing from the manager object. Lazily fetching objects provides two main benefits. First, objects are copied directly from the manager to the log object without passing through the client. Avoiding copying through the client is of particular benefit if the servers that implement the manager object and log object are co-located. Second, lazily fetching objects allows a client to lock a large number of objects (e.g., the range of metadata objects that describe each file under a directory subtree), but only objects that are used are copied.

To enable implicit import, upon locking oid, the manager object stores $\mathsf{vs}_{\mathsf{cid}}$ in addition to cid ($\mathsf{locktable}[\mathsf{oid}] \leftarrow \langle \mathsf{cid}, \mathsf{vs}_{\mathsf{cid}} \rangle$). The value of $\mathsf{vs}_{\mathsf{cid}}$ represents the earliest vs at which an IMPORT is allowed. Upon receiving an APPEND request that touches an object oid that has not been imported, a log server queries the manager object with $\langle \mathsf{vs}, \mathsf{oid} \rangle$ to see if the client has locked the object. If $\mathsf{locktable}[\mathsf{oid}] = \langle \mathsf{cid}, \mathsf{vs}' \rangle$ for some $\mathsf{vs}' \leq \mathsf{vs}$, the manager object and can infer that oid was locked and could have been imported at vs'. Thus, the manager object returns the value of oid, and the log server imports oid. Because NEXT_VS sets $\mathsf{state}[\mathsf{oid}] \leftarrow$ NULL at the log server after REPLAY+UNLOCK sets $\mathsf{locktable}[\mathsf{oid}] \leftarrow$ NULL at the manager object, at most one implicit IMPORT matches each LOCK. To ensure that $\mathsf{vs}'' \leq \mathsf{vs}$ for $\mathsf{locktable}[\mathsf{oid}] = \langle \mathsf{cid}, \mathsf{vs}'' \rangle$ at the manager object and vs at the log server, upon locking an object, the manager returns $\mathsf{vs}_{\mathsf{cid}}$, which the client propagates as argument vs' to APPEND. If $\mathsf{vs}' > \mathsf{vs}$ at the log server, the log server updates vs (line 2300), described in Section B.6.

[4]To avoid circular dependencies, the primary must order operations from log objects immediately, unlike operations from clients in Section A.3.2.

[5]$\log_{\mathsf{cid}}$ need not include $\mathsf{vs}_{\mathsf{cid}}$ because the manager knows $\mathsf{vs}_{\mathsf{cid}}$

B.5 Retrying if APPEND Returns FAILURE

As APPEND is constructed above, a log server may return FAILURE for a request that a different log server includes in its request_log. Thus, if any log server returns FAILURE, the client must determine whether req will be replayed at the manager object or not. The client does so using a new operation at the manager object: $\langle$RETRY, cid, req, reqno, nonce$\rangle$. RETRY requires the manager object to store a new variable, $\text{nonce}_{\text{cid}}$, which is initialized to 0. If req has already been replayed at the manager object, the manager object's response to RETRY tells the client to wait for successful APPEND responses at the log object. If req cannot be replayed at the manager object, the RETRY operation acts like an $\langle$EXEC, cid, req$\rangle$ operation. In order to prevent a RETRY operation from executing an operation that will subsequently be replayed, the primary provides proof to the manager object that it has fetched all the most recent request_logs, and the manager object sets $\text{reqno}_{\text{cid}} \leftarrow \text{reqno}$. The nonce serves as an inexpensive proof that the most recent request_logs have been fetched. The rest of this section steps through an invocation of the RETRY operation.

During an APPEND operation, if any log server returns FAILURE before $2f+1$ log servers return successfully, the client invokes **mgrobj**($\langle$RETRY, cid, req, reqno, nonce$\rangle$) at the manager object, where nonce is an unpredictable random nonce chosen by the client. The RETRY invocation is authenticated like any other **mgrobj**$(\ldots)$ invocation. Before ordering the RETRY operation, a correct primary ensures that either $\text{reqno}_{\text{cid}} \geq \text{reqno}$ or $\text{nonce}_{\text{cid}} = \text{nonce}$, as follows. If $\text{reqno}_{\text{cid}} < \text{reqno}$, the primary invokes $\langle$UNLOCK, vs_{cid}, NULL, nonce$\rangle$ at the log object, which is a special form of UNLOCK that does not add anything to unlocking but passes nonce through to FETCH. Log servers respond by invoking **mgrobj**($\langle$FETCH, cid, vs, NULL, request_log, nonce$\rangle$). The manager object executes REPLAY+UNLOCK upon seeing $2f+1$ request logs from distinct log servers for cid, vs_{cid}, and matching oid and matching nonce, and it sets $\text{nonce}_{\text{cid}} \leftarrow \text{nonce}$. If $\text{reqno}_{\text{cid}} < \text{reqno}$ even after REPLAY+UNLOCK, the primary unlocks objects touched by req, as it would before ordering $\langle$EXEC, cid, req$\rangle$.

The primary then orders $\langle$RETRY, cid, req, reqno, nonce$\rangle$. There are three cases for handling the RETRY at the manager object. First, if $\text{reqno}_{\text{cid}} < \text{reqno}$ and $\text{nonce}_{\text{cid}} \neq \text{nonce}$, or if $\text{reqno}_{\text{cid}} < \text{reqno}$ and $\text{nonce}_{\text{cid}} = \text{nonce}$ but an oid that is touched by req is locked, the primary is faulty and a new primary will be elected. Second, if $\text{reqno}_{\text{cid}} \geq \text{reqno}$, then the request was replayed at the manager object, so the manager object returns vs_{cid}. Upon receiving vs_{cid}, the client infers that the request completed at the log object. The client sets $\text{vs} \leftarrow \text{vs}_{\text{cid}}$ and retries invoking $\langle$APPEND, vs, reqno, req$\rangle$ at the log object. Propagating vs_{cid} in APPEND ensures that each correct log server will issue NEXT_VS (line 2300), which will ensure that reqno completes. Thus, the client can and does wait for $f+1$ such correct log servers to provide matching responses.

Third, if $\text{reqno}_{\text{cid}} < \text{reqno}$ and $\text{nonce}_{\text{cid}} = \text{nonce}$, the manager object sets $\text{reqno}_{\text{cid}} \leftarrow \text{reqno}$, $\text{vs}_{\text{cid}} \leftarrow \text{vs}_{\text{cid}} + 1$, and $\text{log}_{\text{cid}}.\text{request\_log}[\text{reqno}'] \leftarrow$ NO-OP for $\text{reqno}' \in \{\text{reqno}_{\text{cid}}+1, \ldots, \text{reqno}\}$. Because $\text{nonce}_{\text{cid}} = \text{nonce}$, the primary has proven that it fetched the most recent request logs from $2f+1$ log servers after the $\langle$RETRY, cid, $*$, reqno, nonce$\rangle$ operation was invoked. (Assuming the primary could not guess the nonce value, the UNLOCK requests and FETCH responses that included $\text{reqno}_{\text{cid}}$ must have followed the RETRY invocation.) An $\langle$APPEND, $*$, reqno$'$, $*\rangle$ that succeeded at $2f+1$ log servers will overlap on at least one correct log server, so reqno$'$ will be replayed. Thus, the manager object can pad request_log with NO-OP requests, which do nothing other than advance

reqno, such that the log servers will advance to reqno upon the next NEXT_VS. The manager object then executes $\mathsf{retval} \leftarrow$ **execute_request**$(\mathsf{cid}, \mathsf{req}, \mathsf{state})$, as it would when invoking $\langle \text{EXEC}, \mathsf{cid}, \mathsf{req}\rangle$, and returns $\langle \mathsf{vs}_{\mathsf{cid}}, \mathsf{retval}\rangle$. The client sets $\mathsf{vs} \leftarrow \mathsf{vs}_{\mathsf{cid}}$ and returns retval to the application. The manager object returns $\mathsf{vs}_{\mathsf{cid}}$ such that the client can propagate $\mathsf{vs}_{\mathsf{cid}}$ in its next APPEND request, ensuring the log object will execute NEXT_VS and advance to reqno.

B.6 State Transfer and NEXT_VS

If a log server falls behind, such that the value of vs or reqno is less that the values vs' or reqno' sent in APPEND and UNLOCK, then it transfers state from other servers. If $\mathsf{reqno} < \mathsf{reqno}'$ during an $\langle \text{APPEND}, *, \mathsf{reqno}', *\rangle$, the log server fetches requests from other log servers (**missed_reqs**$(\mathsf{reqno}+1)$ on line 2302). If $\mathsf{vs} < \mathsf{vs}'$ during $\langle \text{APPEND}, \mathsf{vs}', *, *\rangle$ or $\langle \text{UNLOCK}, \mathsf{vs}', *\rangle$, then the log server fetches a checkpoint of the state at vs' from other log servers (**get_view**(vs') on lines 2300 and 2400).

B.6.1 missed_reqs(...)

Upon receiving $\langle \text{APPEND}, *, \mathsf{reqno}', *\rangle$, a log server may find that $\mathsf{reqno}' > \mathsf{reqno}+1$ (line 2302) because it missed an APPEND. If the client is correct, it completed APPEND for each of $\mathsf{reqno}+1, \ldots, \mathsf{reqno}'-1$, so $f+1$ correct log servers appended request $\mathsf{reqno}+1$ to their request logs. In **missed_reqs**$(\mathsf{reqno}+1)$, the log server queries each log server for request $\mathsf{reqno}+1$. Upon finding $f+1$ matching responses, the log server replays $\mathsf{reqno}+1$.[6] The log server repeats this process until $\mathsf{reqno}' = \mathsf{reqno}+1$. Of course, in practice, log servers fetch multiple missed requests at once.

If the client is not correct, **missed_reqs**$(\mathsf{reqno}+1)$ may not find $f+1$ matching request log entries for request $\mathsf{reqno}+1$, which will prevent a faulty client from making progress. But, it will not block the primary from completing UNLOCK operations or other clients from completing APPEND operations on their log servers.

B.6.2 NEXT_VS and get_view(...)

This section describes how a log server could fetch checkpoints from the manager object, but it is merely a bridge into Section B.6.3, which describes how to transfer state from other log servers.

The NEXT_VS operation is issued implicitly by propagating $\mathsf{vs}_{\mathsf{cid}}$ as vs' in UNLOCK and APPEND requests. Upon receiving $\langle \text{APPEND}, \hat{\mathsf{vs}}, *, *\rangle$ or $\langle \text{UNLOCK}, \hat{\mathsf{vs}}, *\rangle$, a log server may find that $\hat{\mathsf{vs}} > \mathsf{vs}$ (lines 2300 and 2400), in which case it calls **get_view**$(\hat{\mathsf{vs}})$ to request $\langle \text{NEXT\_VS}, \mathsf{vs}', \mathsf{reqno}', \mathsf{state}'\rangle$ from the manager object. Checkpoint $\langle \mathsf{vs}', \mathsf{reqno}', \mathsf{state}'\rangle$ is generated after REPLAY+UNLOCK by setting $\mathsf{vs}' \leftarrow \mathsf{vs}_{\mathsf{cid}}$ and $\mathsf{reqno}' \leftarrow \mathsf{reqno}_{\mathsf{cid}}$, and processing each oid as follows. If $\mathsf{locktable}[\mathsf{oid}] = \langle \mathsf{cid}, *\rangle$, $\mathsf{state}'[\mathsf{oid}] \leftarrow \mathsf{state}[\mathsf{oid}]$. Otherwise, $\mathsf{state}'[\mathsf{oid}] \leftarrow \text{NULL}$.

Upon receiving a checkpoint $\langle \mathsf{vs}', \mathsf{reqno}', \mathsf{state}'\rangle$, if $\mathsf{vs} \geq \mathsf{vs}'$, the checkpoint is ignored. Otherwise, the log server sets $\mathsf{vs} \leftarrow \mathsf{vs}'$ and does the following. If $\mathsf{reqno}' < \mathsf{reqno}$, then for each oid such that $\mathsf{state}'[\mathsf{oid}] = \text{NULL}$, $\mathsf{state}[\mathsf{oid}] \leftarrow \text{NULL}$ (that is, objects unlocked at the manager object are unlocked at the log server). If $\mathsf{reqno}' \geq \mathsf{reqno}$, then $\mathsf{reqno} \leftarrow \mathsf{reqno}'$ and $\mathsf{state} \leftarrow \mathsf{state}'$.

[6]During replay, an implicit import may fail if an object is unlocked at the manager object, increasing, $\mathsf{vs}_{\mathsf{cid}}$. Thus, if replay fails, the log server calls **get_view**$(\mathsf{vs}+1)$.

B.6.3 Replaying Requests at the Log Object

Rather than replaying the request log at the manager object, Zzyzx replays the request log at the log object, which is often inexpensive because log servers may have already computed the result. This section describes replaying the request log at the log object rather than at the manager object. It does not consider re-using results that the log servers have already computed, which is left for Section B.6.4.

During REPLAY+UNLOCK, the manager object chooses the longest $\mathsf{request\_log}$ and stores it and the oid being unlocked in $\mathsf{log}_{\mathsf{cid}}$, as in Section B.3, but the manager object does not replay $\mathsf{request\_log}$ or unlock oid. Instead, upon choosing $\mathsf{request\_log}$, the primary sends each log server a REPLAY request, which tells the log server to fetch $\langle \mathsf{vs}', \mathsf{oid}, \mathsf{request\_log}' \rangle$ from the manager object.[7] The manager object returns $\mathsf{vs}_{\mathsf{cid}}$ and the values of oid and $\mathsf{request\_log}$ found in $\mathsf{log}_{\mathsf{cid}}$. (The manager object may also be able to provide $\langle \mathsf{vs}', \mathsf{oid}, \mathsf{request\_log}' \rangle$ directly to log servers in its response to FETCH, as in Section 5.3.3.)

Upon receiving $\langle \mathsf{vs}', \mathsf{oid}, \mathsf{request\_log}' \rangle$ from the manager object, a log server does as follows. If $\mathsf{vs} > \mathsf{vs}'$, the request is stale and is ignored. If $\mathsf{vs}' > \mathsf{vs}$, the log server fetches $\mathsf{checkpoint}_{\mathsf{vs}'}$ from $f+1$ other log servers with matching checkpoints.[8] Otherwise, it is the case that $\mathsf{vs} = \mathsf{vs}'$, so the log server lets $\langle \mathsf{state}', \mathsf{reqno}' \rangle \leftarrow \mathsf{checkpoint}_{\mathsf{vs}}$ and replays $\mathsf{request\_log}'$, starting at reqno', on state'.[9] When replay completes, the log server lets $\mathsf{vs} \leftarrow \mathsf{vs}+1$, $\mathsf{obj} \leftarrow \mathsf{state}'[\mathsf{oid}]$ and $\mathsf{state}'[\mathsf{oid}] \leftarrow \text{NULL}$, and it creates a new checkpoint for the new value of vs, $\mathsf{checkpoint}_{\mathsf{vs}} \leftarrow \langle \mathsf{state}', \mathsf{reqno}'' \rangle$, where reqno'' is the last replayed request. The log server sets $\mathsf{state}[\mathsf{oid}] \leftarrow \text{NULL}$, and, if $\mathsf{reqno}'' \geq \mathsf{reqno}$, it sets $\mathsf{state} \leftarrow \mathsf{state}'$ and $\mathsf{reqno} \leftarrow \mathsf{reqno}''$. Finally, the log server sends obj to the manager object, which, upon receiving $2f+1$ matching values, sets $\mathsf{locktable}[\mathsf{oid}] \leftarrow \text{NULL}$, $\mathsf{state}[\mathsf{oid}] \leftarrow \mathsf{obj}$, and $\mathsf{vs}_{\mathsf{cid}} \leftarrow \mathsf{vs}_{\mathsf{cid}} + 1$.

B.6.4 Avoiding Replay

This section considers how to avoid executing requests twice at each log server, now that Section B.6.3 has moved replay to the log object. Two request logs, $\mathsf{request\_log}'$ and $\mathsf{request\_log}$, are said to be consistent after request reqno' if, for $\widehat{\mathsf{reqno}} > \mathsf{reqno}'$, it is the case that $\mathsf{request\_log}'[\widehat{\mathsf{reqno}}] = \mathsf{request\_log}[\widehat{\mathsf{reqno}}]$ when $\mathsf{request\_log}'[\widehat{\mathsf{reqno}}] \neq \text{NULL}$ and $\mathsf{request\_log}[\widehat{\mathsf{reqno}}] \neq \text{NULL}$. Section B.6.3 replays $\mathsf{request\_log}'$ to compute $\mathsf{checkpoint}_{\mathsf{vs}}$ and obj. Suppose at some log server that the request log being replayed to reach the next vs, $\mathsf{request\_log}'$, is consistent with $\mathsf{request\_log}$ after reqno', where $\mathsf{request\_log}$ is the log server's request log and $\mathsf{reqno}' = \mathsf{checkpoint}_{\mathsf{vs}-1}.\mathsf{reqno}$.[10]

Let reqno'' be the last request in $\mathsf{request\_log}'$. If $\mathsf{reqno}'' \geq \mathsf{reqno}$, $\mathsf{request\_log}'$ contains requests missing in $\mathsf{request\_log}$, so the log server executes $\mathsf{reqno}+1, \ldots, \mathsf{reqno}''$ from $\mathsf{request\_log}'$ on state, lets $\mathsf{vs} \leftarrow \mathsf{vs}+1$, $\mathsf{obj} \leftarrow \mathsf{state}[\mathsf{oid}]$, $\mathsf{state}[\mathsf{oid}] \leftarrow \text{NULL}$, and creates $\mathsf{checkpoint}_{\mathsf{vs}} \leftarrow$

[7] Similarly, a newly elected primary sends log servers REPLAY requests for each $\mathsf{log}_{\mathsf{cid}}$ that has not been replayed.

[8] Because $2f+1$ log servers will generate a new checkpoint for vs before the manager object advances $\mathsf{vs}_{\mathsf{cid}}$ to vs, the log server will always find $f+1$ log servers with matching checkpoints. Also, note that standard techniques apply, such as finding $f+1$ log servers with matching hash trees of state before fetching state in chunks.

[9] During replay, an implicit import may fail, but only if $2f+1$ other log servers completed REPLAY, unlocking a required oid at the manager object. Thus, if implicit import fails, the log server can abandon state' and stop replaying.

[10] $\mathsf{request\_log}$ and $\mathsf{request\_log}'$ may be inconsistent if the client is faulty or if RETRY padded $\mathsf{request\_log}'$ with NO-OPS.

$\langle \mathsf{state}, \mathsf{reqno}'' \rangle$. If $\mathsf{reqno}'' < \mathsf{reqno}$, then the log server computes $\mathsf{checkpoint}_{\mathsf{vs}}$ as follows. It computes $\mathsf{checkpoint}_{\hat{\mathsf{reqno}}} \leftarrow \langle \mathsf{state}, \hat{\mathsf{reqno}} \rangle$ upon executing request $\hat{\mathsf{reqno}}$ in $\mathsf{request\_log}$. Then, because executing requests is deterministic, $\mathsf{checkpoint}_{\mathsf{vs}} = \mathsf{checkpoint}_{\mathsf{reqno}''}$. Because executing requests is deterministic, computing $\mathsf{checkpoint}_{\mathsf{reqno}}$ for each request need not be expensive. A checkpoint can be computed from three items: the previous checkpoint, objects that have since been imported, and requests that have since been executed. As in Section 5.4.1, full checkpoints can be computed at fixed intervals to bound replay. Furthermore, if $\mathsf{state}[\mathsf{oid}]$ has not been touched since reqno'', the log server lets $\mathsf{obj} \leftarrow \mathsf{state}[\mathsf{oid}]$, obviating the need for replaying $\mathsf{request\_log}'$.

B.6.5 Garbage Collection

As described, the $\mathsf{request\_log}$ and checkpoints at the log server can grow to consume a substantial amount of memory. This section describes how to reclaim this memory. If $\mathsf{checkpoint}_{\mathsf{vs}} = \langle *, \mathsf{reqno}' \rangle$, then for $\hat{\mathsf{reqno}} \leq \mathsf{reqno}'$ the log server can let $\mathsf{request\_log}[\hat{\mathsf{reqno}}] \leftarrow$ NULL. If $\mathsf{request\_log}[\hat{\mathsf{reqno}}]$ is ever requested by **missed_reqs**$(\hat{\mathsf{reqno}})$ at another log server, this log server can tell the other log server to fetch the checkpoint for vs. Similarly, upon computing a full checkpoint $\mathsf{checkpoint}_{\mathsf{vs}}$, all prior checkpoints can be discarded. If another log server requests $\mathsf{checkpoint}_{\mathsf{vs}'}$ for $\mathsf{vs}' < \mathsf{vs}$, this log server can tell the other log server to fetch the checkpoint for vs.

If $\mathsf{vs}_{\mathsf{cid}}$ is increasing at the manager object, a log server may fetch several checkpoints without finding $f+1$ matching values. Note that this problem will never block UNLOCK, because, if it does, $\mathsf{vs}_{\mathsf{cid}}$ will stop increasing at the manager object, such that the log server will eventually find $f+1$ matching checkpoint. A log server may find several non-matching checkpoints, however, during an APPEND. If APPEND is blocked, $\mathsf{vs}_{\mathsf{cid}}$ may increase, but $\mathsf{reqno}_{\mathsf{cid}}$ will not increase. Thus, checkpoints from correct log servers will be the same except that checkpoints for higher vs values will have fewer locked objects. More formally, $\mathsf{checkpoint}_{\mathsf{vs}} = \langle \mathsf{state}, \mathsf{reqno} \rangle$ and $\mathsf{checkpoint}_{\mathsf{vs}'} = \langle \mathsf{state}', \mathsf{reqno}' \rangle$ are said to be consistent if $\mathsf{reqno} = \mathsf{reqno}'$ and for each oid such that $\mathsf{state}[\mathsf{oid}] \neq$ NULL and $\mathsf{state}'[\mathsf{oid}] \neq$ NULL, it is the case that $\mathsf{state}[\mathsf{oid}] = \mathsf{state}'[\mathsf{oid}]$.

Upon receiving $f+1$ consistent checkpoints, a log server queries the manager object for $\mathsf{vs}_{\mathsf{cid}}$ and a list of oids locked by cid. If $\mathsf{reqno}_{\mathsf{cid}}$ matches the value of reqno in each checkpoint, $\mathsf{vs}_{\mathsf{cid}}$ is greater than or equal to the value of vs for each checkpoint, and if, for each oid locked by cid, the value of oid matches in each checkpoint, the client constructs a checkpoint for $\mathsf{vs} \leftarrow \mathsf{vs}_{\mathsf{cid}}$ as follows: $\mathsf{checkpoint}_{\mathsf{vs}} \leftarrow \langle \mathsf{reqno}', \mathsf{state}' \rangle$, where reqno' is the matching value in each checkpoint, $\mathsf{state}'[\mathsf{oid}]$ is the matching value for locked oids, and $\mathsf{state}'[\mathsf{oid}] \leftarrow$ NULL for unlocked oids. A log server will eventually find $f+1$ such correct log servers and advance to vs, which will allow the APPEND operation to continue.

B.7 Using MACs Instead of Signatures in APPEND

The pseudocode in Figure B.3.1 requires that APPEND requests are signed, such that the manager object can authenticate request logs when choosing the longest $\mathsf{request\_log}$ to store in $\mathsf{log}_{\mathsf{cid}}$. This section describes how to avoid signatures. Suppose the client replaced the signature with an authenticator, which is a vector of MACs, one for each log server. The authenticator will allow each log server to verify the authenticity of an APPEND request, but it cannot also be verified by the manager object. If the longest request_log is proposed by $f+1$ log servers, the manager can trust that it was authenticated (at least one correct log server proposed request_log and authenticated each request), but the manager must also be able to choose a request log when fewer than $f+1$ match. To do so, log servers implement the following voting scheme.

To invoke APPEND, the client generates a *layered authenticator*, which is a second vector of MACs computed over the authenticator. The outer vector of MACs is the outer authenticator, while the inner vector is the inner authenticator. Upon receiving an APPEND request, a correct log server authenticates the request and the inner authenticator using the outer authenticator. The outer authenticator is discarded, but the inner authenticator is appended to request_log in place of the signature. Upon receiving UNLOCK, each log server sends its authenticated request_log to the primary,[11] which accumulates a set of $2f+1$ request_logs that it sends to each log server. Upon receiving authenticated request_logs from $2f+1$ distinct log servers, each log server tests whether the inner MACs properly authenticate the request in a request_log. Each log server votes on the request_logs by invoking FETCH at the manager object, voting in favor for each request_log where all requests are properly authenticated, and voting against each request_log where any request is not properly authenticated. The manager object chooses the longest request_log with $f+1$ votes in favor, or the empty log if no request_log achieves $f+1$ votes.

Further optimization: The outer authenticator is used by the log server to ensure that a request is authentic before executing it and appending it to the request log, but the isolation property discussed in Section B.1 enables the client to avoid generating the outer authenticator in mostly benign environments. If the client does not generate an outer authenticator, the log server can include the entire inner authenticator in its authenticated response to the client. If the inner authenticator is corrupted during invocation, the MAC in the response will not check, so the client will generate the outer authenticator and try again. Upon receiving different requests for the same reqno, the log server verifies the MAC in the outer authenticator, possibly undoing operation reqno. In its next request, the client includes the prior inner authenticator or its hash, such that the log server can authenticate the prior inner authenticator.

Upon receiving authenticated responses from $2f+1$ log servers, a correct client can infer that at least 1 correct log server that has appended req and the inner authenticator to its request log will participate in the next FETCH. Thus, if each reqno is voted on independently (rather than as part of a request_log), at least $f+1$ correct log servers will vote in favor for the request, so the request will be reflected in the next REPLAY+UNLOCK. Similarly, if a correct log server calls **missed_reqs**(reqno), then $f+1$ correct log servers will return req.

Even further optimization: The log server need not authenticate the prior inner authenticator until FETCH or **missed_reqs**(...). Thus, if the client sends a cumulative MAC as the inner authenticator of the entire request_log (or its hash), only the cumulative MAC need be verified. Thus, the lower bound replicated state machine bottleneck MACs per request in Figure 5.1.2 is 1, just a single MAC. This technique is primarily of theoretical interest. To prevent a long sequence of speculative requests (and potential undos), the log server may wish to verify the inner MAC before executing a request, or at least every few requests. The implementation evaluated in Section 5.5 verifies the inner MAC for each request.

[11] request_log could be signed, or log servers could generate authenticators, using signatures on failure.

Bibliography

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 59–72. USENIX Association, 2005.

[2] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74. ACM Press, 2005.

[3] M. Abd-El-Malek, G. R. Ganger, M. K. Reiter, J. J. Wylie, and G. R. Goodson. Lazy verification in fault-tolerant distributed storage systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 179–190. IEEE Computer Society, 2005.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14. USENIX Association, 2002.

[5] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 336–345. IEEE Computer Society, 2005.

[6] M. K. Aguilera and R. Swaminathan. Brief Announcement: Remote storage with Byzantine servers. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, pages 312–313. ACM Press, 2007.

[7] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 197–206. IEEE Computer Society, 2008.

[8] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–126. ACM Press, 1995.

[9] L. N. Bairavasundaram, G. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *Proceedings of the 6th*

USENIX Conference on File and Storage Technologies, pages 223–238. USENIX Association, 2008.

[10] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212. ACM Press, 1991.

[11] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology – CRYPTO '94*, pages 216–233. Springer-Verlag, 1994.

[12] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73. ACM Press, 1993.

[13] W. J. Bolosky, J. R. Douceur, and J. Howell. The Farsite project: a retrospective. *SIGOPS Operating Systems Review*, 41(2):17–26, 2007.

[14] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The Zettabyte File System. Technical report, Sun Microsystems.

[15] A. Z. Broder. Some applications of Rabin's fingerprinting method. *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152, 1993.

[16] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association, 2006.

[17] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantipole: practical asynchronous Byzantine agreement using cryptography (extended abstract). In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132. ACM Press, 2000.

[18] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 191–202. IEEE Press, 2005.

[19] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 115–124. IEEE Computer Society, 2006.

[20] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the 9th ACM Symposium on Theory of Computing*, pages 106–112. ACM Press, 1977.

[21] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT Laboratory for Computer Science, January 2001. Technical Report MIT/LCS/TR-817.

[22] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 298–313. ACM Press, 2003.

[23] M. Castro and B. Liskov. Authenticated Byzantine fault tolerance without public-key cryptography. Technical Memo MIT-LCS-TM-589, MIT, June 1999.

[24] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[25] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM Press, 2007.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–28. USENIX Association, 2006.

[27] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[28] G. Chockler, R. Guerraoui, and I. Keidar. Amnesic distributed storage. In *Proceedings of the 21st International Symposium on Distributed Computing*, pages 139–151. Springer-Verlag, 2007.

[29] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Proceedings of the 26th Annual Symposium on the Foundations of Computer Science*, pages 383–395. IEEE Press, 1985.

[30] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2009.

[31] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association, 2004.

[32] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 177–190. USENIX Association, 2006.

[33] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics: General results and applications. In *Advances in Cryptology – ASIACRYPT*, pages 1–20. Springer-Verlag, 2006.

[34] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 205–220. ACM Press, 2007.

[35] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 321–334. USENIX Association, 2006.

[36] A. Doudou, R. Guerraoui, and B. Garbinato. Abstractions for devising Byzantine-resilient state machine replication. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 144–153. IEEE Computer Society, 2000.

[37] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[38] D. Ellard and M. Seltzer. New NFS tracing tools and techniques for system analysis. In *Proceedings of the 17th USENIX Large Installation System Administration Conference*, pages 73–86. USENIX Association, 2003.

[39] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on the Foundations of Computer Science*, pages 427–437. IEEE Press, 1987.

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

[41] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43. ACM Press, 2003.

[42] B. Gladman. SHA1, SHA2, HMAC and key derivation in C.
http://fp.gladman.plus.com/cryptography_technology/sha.

[43] L. Gong. Securely replicating authentication services. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 85–91. IEEE Computer Society, 1989.

[44] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–144. IEEE Computer Society, 2004.

[45] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. The safety and liveness properties of a protocol family for versatile survivable storage infrastructures. Technical Report CMU–PDL–03–105, Parallel Data Laboratory, Carnegie Mellon University, 2004.

[46] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 175–188. ACM Press, 2007.

[47] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[48] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 73–86. ACM Press, 2007.

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

[50] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE Computer Society, 2003.

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

[52] J. Katcher. Postmark: a new file system benchmark. Technical report TR3022, Network Appliance, October 1997.

[53] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems*, pages 61–75, 1997.

[54] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, pages 317–326. IEEE Computer Society, 1998.

[55] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 45–58. ACM Press, 2007.

[56] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. Technical Report TR-07-40, The University of Texas at Austin, Department of Computer Sciences, August 2007.

[57] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 575–584. IEEE Computer Society, 2004.

[58] H. Krawczyk. Distributed fingerprints and secure information dispersal. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 207–218. ACM Press, 1993.

[59] H. Krawczyk. LFSR-based hashing and authentication. In *Advances in Cryptology – CRYPTO '94*, pages 129–139. Springer-Verlag, 1994.

[60] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity lost and parity regained. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 127–141. USENIX Association, 2008.

[61] M. N. Krohn, M. J. Freedman, and D. Mazieres. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Press, 2004.

[62] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201. ACM Press, 2000.

[63] K. Kursawe. Optimistic Byzantine agreement. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 262–267. IEEE Computer Society, 2002.

[64] L. Lamport. Paxos made simple. *ACM SIGACT News*, pages 18–25, Dec 2001.

[65] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[66] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proceedings of the 26th International Conference on Distributed Computing Systems*, pages 34–43. IEEE Computer Society, 2006.

[67] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th IEEE workshop on Computer Security Foundations*, pages 116–124. IEEE Computer Society, 1997.

[68] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):201–213, 1998.

[69] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11:203–213, 1998.

[70] D. Malkhi, M. Reiter, and N. Lynch. A correctness condition for memory shared by byzantine processes. `http://groups.csail.mit.edu/tds/papers/Lynch/lynch-malk-reit.html` as retrieved on 16 March 2008.

[71] D. Malkhi, M. K. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. *SIAM Journal of Computing*, 29(6):1889–1906, 2000.

[72] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 369–384. USENIX Association, 2008.

[73] J.-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–388. IEEE Computer Society, 2002.

[74] P. Maymounkov. Online codes. Technical Report TR2002–833, Secure Computer Systems Group, New York University, 2002.

[75] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21(4):339–374, 1984.

[76] N. Möller. *Nettle Manual*, 1.15 edition, 2006.

[77] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the ACM/IEEE SC2004 Conference*, page 53. IEEE Computer Society, 2004.

[78] M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and KDCs. In *Advances in Cryptology – EUROCRYPT '99*, pages 327–346. Springer-Verlag, 1999.

[79] W. Nevelsteen and B. Preneel. Software performance of universal hash functions. In *Advances in Cryptology – EUROCRYPT '99*, pages 24–41. Springer-Verlag, 1999.

[80] T. Pedersen. Distributed provers with applications to undeniable signatures. In *Advances in Cryptology – EUROCRYPT '91*, pages 221–242. Springer-Verlag, 1991.

[81] J. S. Plank. The RAID-6 liberation codes. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association, 2008.

[82] R. Primmer and C. D. Halluin. Collision and preimage resistance of the Centera content address. Technical report, EMC^2 Corporation, 2005.

[83] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[84] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[85] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM Journal of Applied Mathematics*, 8:300–304, 1960.

[86] M. K. Reiter. Secure agreement protocols: reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80. ACM Press, 1994.

[87] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 99–110. Springer-Verlag, 1995.

[88] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 15–28. ACM Press, 2001.

[89] R. Rodrigues, P. Kouznetsov, and B. Bhattacharjee. Large-scale Byzantine fault tolerance: Safe but not always live. In *Proceedings of the 3rd Workshop on Hot Topics in System Dependability*. USENIX Association, 2007.

[90] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Proceedings of the 11th International Workshop on Fast Software Encryption*. Springer-Verlag, 2004.

[91] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58. ACM Press, 2004.

[92] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 231–244. USENIX Association, 2002.

[93] T. Schwarz. Verification of parity data in large scale storage systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 2004.

[94] V. Shoup. On fast and provably secure message authentication based on universal hashing. In *Advances in Cryptology – CRYPTO '96*, pages 313–328. Springer-Verlag, 1996.

[95] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. Conflict-free quorum based BFT protocols. Technical Report TR-2007-2, Max Planck Institute for Software Systems, August 2007.

[96] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 189–204. USENIX Association, 2008.

[97] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58. USENIX Association, 2003.

[98] M. A. Stadler. Publicly verifiable secret sharing. In *Advances in Cryptology – EUROCRYPT '96*, pages 190–199. Springer-Verlag, 1996.

[99] Sun Microsystems. *ZFS On-Disk Specification Draft*, 2006.

[100] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. In *Proceedings of the 3rd International Conference on Autonomic Computing*, pages 187–198. IEEE Computer Society, 2006.

[101] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, 1984.

[102] D. Travis. On irreducible polynomials in Galois fields. *The American Mathematical Monthly*, 70(10):1089–1090, 1963.

[103] D. Wagner. A generalized birthday problem. In *Advances in Cryptology – CRYPTO '02*, pages 288–304. Springer-Verlag, 2002.

[104] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Advances in Cryptology – EUROCRYPT '05*, pages 19–35. Springer-Verlag, 2005.

[105] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. In *International Workshop on Peer-to-Peer Systems*, pages 328–337. Springer-Verlag, 2002.

[106] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[107] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267. ACM Press, 2003.

[108] Z. Zhang, S. Lin, Q. Lian, and C. Jin. RepStore: A self-managing and self-tuning storage backend with smart bricks. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 122–129. IEEE Computer Society, 2004.